

# **Algorithm Analysis And Design**

**Mahesh sathvika,  
IIIT Hyderabad,  
Cse,  
2020101087.**

# CONTENTS

## Greedy Algorithms

- General Explanation
- Exact Definition, Advantages, Disadvantages, When to use greedy
- Example for where greedy fails
- Job Sequencing problem
- Graph Coloring
- Monoplay

## Dynamic Programming

- Definition
  - Memoization
  - Tabulation
- When to use Dynamic programming
  - Overlapping subproblems property
  - Optimal substructure property
- Finding Fibonacci using naive, Tabulation, Memoization
  - Graph for above methods (Fibonacci number using recursion and Tabulation)
- Analysing the graph (about how dynamic runs fast)
- Longest Common subsequence (using naive and Dynamic)
  - Solved using Dynamic programming and naive
  - Analysing the naive and dynamic programming time complexity
- Word wrap problem
  - Explaining why greedy always won't give optimal solution
  - Solving using Dynamic programming

-----For all the problems ,codes and their outputs also provided in this book.-----

## Greedy algorithms

### Definition

#### General explanation:

John and Alex are best friends in their class, John has different types of pens which are of cost 20, 10, 5.

Alex : John i will give u 35 rupees, can u give me your pens in such a way that I should get less number of pens

Jhon : Yes, definitely Alex. (John gave one 20Rs + one 10Rs pen + one 5Rs pen)

So John has many possibilities to give pens like,

$$5+5+5+5+5+5+5,$$

$$10+10+10+5,$$

$$20+10+5$$

To give minimum number of pens, John should 1st give highest price pen (which is 20RS), so we have

$$20+X=35,$$

$$X=15$$

Now, he can give one 10 Rs pen and one 5Rs pen

So, by this John gave a minimum number of pens to Alex (This may not be true every time, but this is to understand easily).

This is how greedy work's.

#### Exact Definition:

Problem solving approach of making the locally optimal choice at each stage with the hope of finding a global optimum. At every stage we choose a local optimal solution that leads us to give global optimal solutions.

#### Advantages :

- Simple
- easy to implement code
- Runs fast

### When we can use Greedy algorithm

#### Greedy choice Property:

A global optimum can be reached by selecting local optimums.

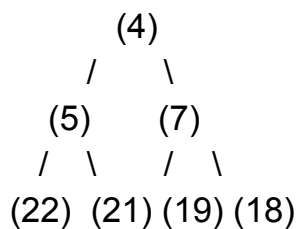
#### Optimal Substructure Property:

A problem follows optimal substructure property if the optimal solution for the problem can be formed on the basis of the optimal solution to its subproblems.

#### Disadvantages:

- In some cases it may not give global optimum solution

Ex:



In the above tree ,if we were asked to find out the maximum number which can be done by adding the numbers in every level such that only one element is taken from each level.

4->7->19=30

by taking local optimum every time ,but we can get better than this

4->5->22=31

So,sometimes greed won't give optimal solutions but it runs fast.

## Job sequencing problem using Greedy method

### Problem Statement:

Given an array of jobs where every job has a deadline and associated profit if the job is done before the given deadline.it is also given that every job takes a single unit of time like(0-1,1-2,...),so the minimum possible deadline for any job is 1.How to maximize the total profit if only one job can be scheduled at a time.

### General Solution:

We can make all possible subsets of a given set of jobs,and find which can give a maximum profit.But,for this time complexity of solution is exponential.

### Using Greedy Algorithm:

#### Steps to solve:

- 1) Sort all the jobs in decreasing order based on their profit
- 2) Iterate through the sorted array and for each job
  - a)Find a time slot j,such that it is empty and j<deadline of job and this should be greatest.Then put the job in this slot and mark this slot is filled
  - b)if no such j exists ,then just do nothing.

```

// Program to find the maximum profit job sequence from a given array
// of jobs with deadlines and profits
#include<iostream>
#include<algorithm>
using namespace std;
// A structure to represent a job
struct Job
{

```

```

    char id;        // Job Id
    int dead;       // Deadline of job
    int profit;     // Profit if job is over before or on deadline
};

// This function is used for sorting all jobs according to profit
bool comparison(Job a, Job b)
{
    return (a.profit > b.profit);
}

// Returns minimum number of platforms required
void printJobScheduling(Job arr[], int n)
{
    // Sort all jobs according to decreasing order of profit
    sort(arr, arr+n, comparison);
    int result[n]; // To store result (Sequence of jobs)
    bool slot[n];  // To keep track of free time slots
    // Initialize all slots to be free
    for (int i=0; i<n; i++)
        slot[i] = false;
    // Iterate through all given jobs
    for (int i=0; i<n; i++)
    {
        // Find a free slot for this job (Note that we start
        // from the last possible slot)
        for (int j=min(n, arr[i].dead)-1; j>=0; j--)
        {
            // Free slot found
            if (slot[j]==false)
            {
                result[j] = i; // Add this job to result
                slot[j] = true; // Make this slot occupied
                break;
            }
        }
    }
    int p=0;
    // Print the result
    for (int i=0; i<n; i++)
        if (slot[i]){
            cout << arr[result[i]].id << " ";
            p=p+arr[result[i]].profit;
        }
    cout << "\nprofit : ";

```

```

    cout << p << " ";
}

int main()
{
    Job arr[] = { {'a', 3, 100}, {'b', 1, 19}, {'c', 2, 27},
                  {'d', 1, 25}, {'e', 4, 15}};
    int n = sizeof(arr)/sizeof(arr[0]);
    cout << "Following is maximum profit sequence of jobs \n";

    printJobScheduling(arr, n);
    return 0;
}
Following is maximum profit sequence of jobs
d c a e
profit : 167

```

The screenshot shows a code editor with the same C++ code. Below the editor is a terminal window with the following output:

```

dell@sathvika-inspiron-14-5408:~/Aad$ g++ jobsequencing.cpp
dell@sathvika-inspiron-14-5408:~/Aad$ ./a.out
Following is maximum profit sequence of jobs
d c a e
profit : 167 dell@sathvika-inspiron-14-5408:~/Aad$

```

**Explanation:**

**Input array of struct:**

NUmber of Jobs	JoB ID	Deadline	Profit
1	a	3	100
2	b	1	19
3	c	2	27
4	d	1	25
5	e	4	15

So,Is we sort the array based on their profits in descending order

**Array after sorting:**

Job ID	Deadline	Profit
<b>a</b>	<b>3</b>	<b>100</b>
<b>c</b>	<b>2</b>	<b>27</b>
<b>d</b>	<b>1</b>	<b>25</b>
<b>b</b>	<b>1</b>	<b>19</b>
<b>e</b>	<b>4</b>	<b>15</b>

So,the job with the highest profit is 100 .we include this to our answer.

‘A’ job has a deadline 3,so it can be done anytime from 0-3.we have possibilities like 0-1,1-2,2-3.but we do job ‘a’ between 2-3,because then we can do other jobs between 0-2 and get more profit.

0-1	1-2	2-3	3-4	4-5
		Job a (100)		

Then we have job c,for job ‘c ‘ the deadline is 2 units of time so we can do this in 0-1 or 1-2.But we do this in 1-2 because then we can do another job in 0-1 time and increase the profit.

0-1	1-2	2-3	3-4	4-5
	Job c (27)	Job a (100)		

Next we have job d,

Which has a deadline of 1 unit,so it should be done between 0-1

0-1	1-2	2-3	3-4	
Job d (25)	Job c (27)	Job a (100)		

Next we have job b,but its deadline is 1 ,so we can’t include this in our required output.

Next we have job e,

Whose deadline is 4,so we can do this job in 3-4

0-1	1-2	2-3	3-4
Job d (25)	Job c (27)	Job a (100)	Job e (15)

So our total profit is  $25+27+100+15=167$

And the sequence in which jobs gonna finish is d,c,a,e

**Time complexity:  $O(n^2)$**

## Graph coloring using Greedy

Graph coloring problem is to assign colors to certain elements of a graph subject to certain constraints. **Vertex coloring** is the most common graph coloring problem. The problem is, given  $m$  colors, find a way of coloring the vertices of a graph such that no two adjacent vertices are colored using the same color. The other graph coloring problems like **Edge Coloring** (No vertex is incident to two edges of same color) and **Face Coloring** (Geographical Map Coloring) can be transformed into vertex coloring

Basically we have three problems in graph coloring,

1.  $m$ -Coloring Decision Problem
2.  $m$ -Coloring permutation problem
3.  $m$ -Coloring Optimization problem

Graph coloring problem is to assign colors to certain elements of a graph subject to

### **1.m-Coloring Decision problem**

The problem is to find if it is possible to assign nodes with  $m$  different colors,such that no two adjacent vertices of the graph are of the same colors.

### **2.m-coloring permutation problems**

If the undirected graph can be colored such a way that,no two adjacent vertices has same color then we have to find in how many ways we can color the graph

### **3.m-coloring Optimization solution**

This is to find the minimum number of colors required to color a graph in such a way that no two adjacent vertices are of same color.This is also called as Chromatic number



So now we use greedy algorithms to color a graph with a minimum number of colors.

### Coloring graph using Greedy Algorithm

Greedy algorithm is very fast but sometimes greedy algorithm don't give an optimal solution to minimal color in the graph. Because it depends on the order in which we are taking the vertices.

1. select any order of vertices in which we are going to color

2. Color first vertex with first color.

3. Do the following for remaining  $V-1$  vertices.

a) Consider the currently picked vertex and color it with the lowest numbered color that has not been used on any previously colored vertices adjacent to it. If all previously used colors appear on vertices adjacent to  $v$ , assign a new color to it.

```
# Python3 program to implement greedy
# algorithm for graph coloring
def addEdge(adj, v, w):

    adj[v].append(w)

    # Note: the graph is undirected
    adj[w].append(v)
    return adj

# Assigns colors (starting from 0) to all
# vertices and prints the assignment of colors
def greedyColoring(adj, V):

    result = [-1] * V
    # Assign the first color to first vertex
    result[0] = 0
    # A temporary array to store the available colors.
    # True value of available[cr] would mean that the
    # color cr is assigned to one of its adjacent vertices
    available = [False] * V
    # Assign colors to remaining V-1 vertices
    for u in range(1, V):

        # Process all adjacent vertices and
        # flag their colors as unavailable
        for i in adj[u]:
            if (result[i] != -1):
```

```

        available[result[i]] = True
    # Find the first available color
    cr = 0
    while cr < V:
        if (available[cr] == False):
            break

        cr += 1

    # Assign the found color
    result[u] = cr
    # Reset the values back to false
    # for the next iteration
    for i in adj[u]:
        if (result[i] != -1):
            available[result[i]] = False
    # Print the result
    for u in range(V):
        print("Vertex", u, " --->  Color", result[u])
# Driver Code
if __name__ == '__main__':

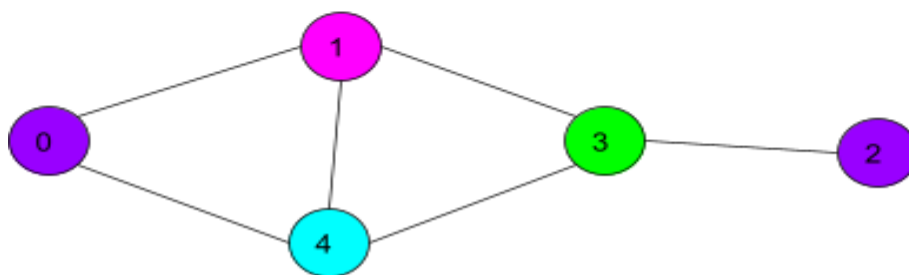
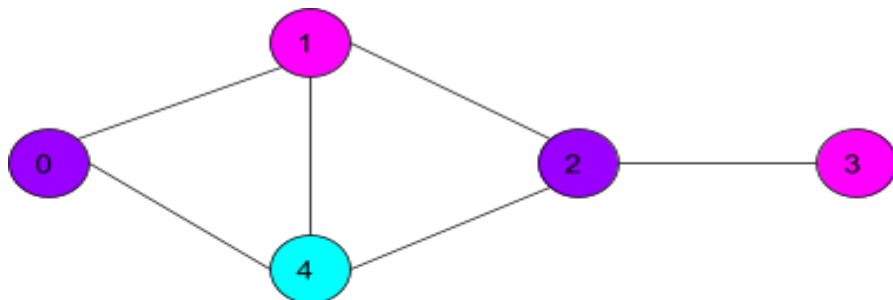
    g1 = [[] for i in range(5)]
    g1 = addEdge(g1, 0, 1)
    g1 = addEdge(g1, 0, 2)
    g1 = addEdge(g1, 1, 2)
    g1 = addEdge(g1, 1, 3)
    g1 = addEdge(g1, 2, 3)
    g1 = addEdge(g1, 3, 4)
    print("Coloring of graph 1 ")
    greedyColoring(g1, 5)
    g2 = [[] for i in range(5)]
    g2 = addEdge(g2, 0, 1)
    g2 = addEdge(g2, 0, 2)
    g2 = addEdge(g2, 1, 2)
    g2 = addEdge(g2, 1, 4)
    g2 = addEdge(g2, 2, 4)
    g2 = addEdge(g2, 4, 3)
    print("\nColoring of graph 2")
    greedyColoring(g2, 5)
# This code is contributed by mohit kumar 29

```

But we always don't get the optimal solution, because it depends on the order in which we are coloring the vertices.

If we take 0,1,2,3,4 as ordering in the 1st graph we can do that with a minimum of three colors, but in the second it is done with four colors.

So, greedy always don't give optimal solution that depends on how we choose the ordering



**Monopoly Game:**

Monopoly is a game played using cards which have values of different denominations like 500,100,50,20,10,5,1.if you are a banker in a monopoly game ,one player asked u to give some value of cards in such a way that u should give a minimum number of cards.

### Using Greedy Algorithm:

- 1.sort the available denominations in decreasing order
- 2.check if we can give that value using available denominations
- 3.Print the minimum number of cards

```
#include<iostream>
#include<algorithm>
using namespace std;
// function used to sort the denominations
bool comparison(int a, int b)
{
    return (a > b);
}
void monopoly(int arr[],int n,int x)
{
    // sort all denominations in decreasing order
    sort(arr,arr+n,comparison);
    int result[n];
    for(int i=0;i<n;i++)
    {
        if(x/arr[i]!=0)
        {
            result[i]=x/arr[i];
            x=x%arr[i];
        }
        else
        {
            result[i]=0;
        }
    }
    int s=0;
    for(int i=0;i<n;i++)
    {
        if(result[i]!=0)
        {
            s=s+result[i];
        }
    }
    cout << arr[i] << " ";
    cout << "---->";
}
```

```

        cout << result[i] << " \n";
    }
    cout << "Minimum number of cards ";
    cout << s << " \n";
}

int main()
{
    int arr[] = {100,500,20,10,50,5,1};
    int n = sizeof(arr)/sizeof(arr[0]);
    int x=534;

    monoplay(arr, n,x);
    return 0;
}

```

```

1 ---->4
dell@sathvika-inspiron-14-5408:~/Aad$ g++ monoplay.cpp
dell@sathvika-inspiron-14-5408:~/Aad$ ./a.out
500 ---->1
100 ---->0
50 ---->0
20 ---->1
10 ---->1
5 ---->0
1 ---->4
Minimum number of cards 7
dell@sathvika-inspiron-14-5408:~/Aad$

```

So let us consider we have {500,100,50,20,10,5,1} as denominations, and the player asks 534 Rs with the minimum number of cards from the banker.

We can give 534 using one 500, then we have 34. Using 100, 50 we can't give so we can give one 20 then we have 14. we can give one 10 and 4 1's.

$$534 = 500 + 20 + 10 + 1 + 1 + 1 + 1$$

In code we have denominations as 100,500,20,10,50,5,1. then using sort function we sort the given array and the output is sorted in decreasing order.

Sort denominations---> 500,100,50,20,10,5,1.

Iterations	Denominations	Number of cards using this denomination
1	500	534/500=1 so we can

		use 1 card of 500 $534 - 500 * 1 = 34$
2	100	$34 / 100 = 0$ so we don't give 100 denomination cards. $34 - 100 * 0 = 34$
3	50	$34 / 50 = 0$ so we don't give 50 denomination cards. $34 - 50 * 0 = 34$
4	20	$34 / 20 = 1$ so we give one 20 denomination card. $34 - 20 * 1 = 14$
5	10	$14 / 10 = 1$ we give one 10 denomination cards. $14 - 10 * 1 = 4$
6	5	$4 / 5 = 0$ so we don't give this card

Finally we give four 1 denomination cards.

500 ---->1

100 ---->0

50 ---->0

20 ---->1

10 ---->1

5 ---->0

1 ---->4

We add all the non-zero cards  $1 + 1 + 1 + 4 = 7$

Minimum number of cards= 7

**Time complexity:**  $O(n \log n)$

**$O(n)$  if given denominations are in sorted order.**

## Dynamic programming

Dynamic programming is mainly optimization over plain recursion. whenever there is a requirement of repeated calls for same input in recursive solution we can optimize this using Dynamic programming. The main idea is to store the results of subproblems, so that we don't re-compute them when they are

needed later. This simple optimization reduces time complexities from exponential to polynomial.

There are two ways we can store the results of subproblems.

- Tabulation

- Memoization

There Are two main properties of a problem that suggest that the given problem can be solved using Dynamic programming:

- Overlapping subproblems

- Optimal substructure property

### 1. Overlapping subproblems

Dynamic Programming is mainly used when solutions of the same subproblems are needed again and again. In dynamic programming, computed solutions to subproblems are stored in a table so that these don't have to be recomputed. So Dynamic Programming is not useful when there are no common (overlapping) subproblems because there is no point storing the solutions if they are not needed again.

Ex: Fibonacci

### 2. Optimal substructure property

A given problem has Optimal Substructure Property if optimal solution of the given problem can be obtained by using optimal solutions of its subproblems.

### Tabulation-Bottom up:

The name itself suggests that we start from bottom and reach to our desired output.

Let us take an example of calculating the factorial of a number .

```
#include <stdio.h>
int main()
{
    // Tabulated version to find factorial x.
    int dp[100];
    int n=6;
    // base case
    dp[0] = 1;
    for (int i = 1; i <=n; i++)
    {
        dp[i] = dp[i - 1] * i;
    }
    printf("%d\n", dp[n]);
}
```

In the above code we start by giving dp[1]=1 and then finding dp[2] using dp[1] and finding dp[3],dp[4] and so on.. Until we get our desired factorial(factorial of 6 in above code).so we start from the bottom most case and we go to our required case.In every iteration we are storing the values in dp[] and using them when they are required.

**Sequence followed** :dp[1]->dp[2]->dp[3]----dp[6].

### **Memoization-Top down:**

Here, we start our journey from the top most destination state and compute its answer by taking in count the values of states that can reach the destination state, till we reach the bottom most base state.

```
#include <stdio.h>
int dp[100]={0};
int solve (int x)
{
    if (x == 0)
        return 1;
    if (dp[x] != 0)
        return dp[x];
    return (dp[x] = x * solve(x - 1));
}
int main()
{
    int p=solve(3);
    printf("%d\n",p);
}
```

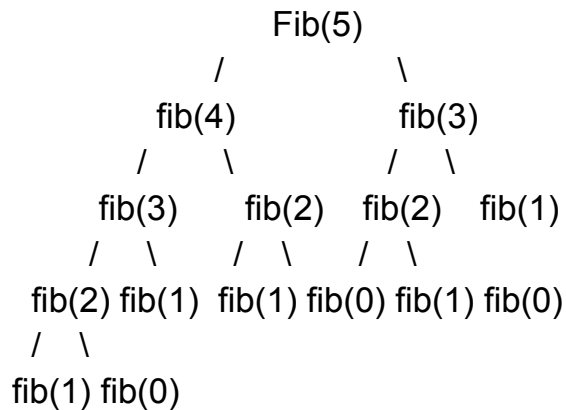
### **Difference between Tabulation and memoization**

	memoization	Tabulation
<b>code</b>	<b>Code is easy to implement</b>	<b>Code get complicated when we have lot of conditions</b>
<b>Speed</b>	<b>Slow,due to lot of recursive calls and return statements</b>	<b>Fast,because we directly access the previous values from the table</b>
<b>Way of solving</b>	<b>Solve from top to bottom state</b>	<b>Solve from bottom to up state</b>



## Finding nth fibonacci number using dynamic programming

We can find the nth fibonacci number using recursion. but ,in recursion we repeatedly re-compute the values again and again which tends to be done in exponential time.



So, in the above tree we are re-computing fib(3), fib(2), fib(1) repeatedly. To overcome this problem we use dynamic programming by storing the subproblems outputs.

## Finding nth fibonacci using recursion

```
#include <stdio.h>
#include <time.h>
int fib(int n)
{
    if (n <= 1)
        return n;
    return fib(n - 1) + fib(n - 2);
}

int main()
{
    int n = 40;

    clock_t begin, end;

    double time_spent;

    begin = clock();

    printf("%d\n", fib(n));

    end = clock();

    time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
```

```

printf("\nTime Taken %lf ", time_spent);

return 0;
}

```

**Time taken to compute this is:0.639040s**

So we observed that we are re-computing the same inputs multiple times,so for every subproblem we store the outputs and we reuse them further without computing again in dynamic programming.

**Why we can do this with Dynamic programming:**

Because,we have overlapping subproblems ,so we can do this problem with dynamic programming

**Dynamic programming using Tabulation:(Iterative version)**

In this ,1st we start from the base case and reach our desired state.

After every iteration we store the output of subproblems in a table,and we use them in the next iteration.

```

#include <stdio.h>
#include <time.h>

int fib(int n)
{
    int f[n + 1];
    int i;
    f[0] = 0;
    f[1] = 1;
    for (i = 2; i <= n; i++)
        f[i] = f[i - 1] + f[i - 2];

    return f[n];
}

int main()
{
    int n = 40;

    clock_t begin, end;
    double time_spent;

    begin = clock(); // Time before calculating Fib number

```

```

printf("Fibonacci number is %d \n", fib(n));

end = clock(); // Time before calculating Fib number

time_spent = (double)(end - begin) / CLOCKS_PER_SEC;

printf("\nTime Taken %lf ", time_spent);

return 0;
}

```

Time required to compute this is :0.000138

### Dynamic programming using memoization:

We start from the nth state and move to the bottom state .

```

#include <stdio.h>

#include <time.h>

#define NIL -1
#define MAX 100

int lookup[MAX];

/* Function to initialize NIL values in lookup table */
void _initialize()
{
    int i;
    for (i = 0; i < MAX; i++)
        lookup[i] = NIL;
}

/* function for nth Fibonacci number */
int fib(int n)
{
    if (lookup[n] == NIL)
    {
        if (n <= 1)
            lookup[n] = n;
        else
            lookup[n] = fib(n - 1) + fib(n - 2);
    }
}

```

```

        return lookup[n];
    }

int main()
{
    int n = 40;

    clock_t begin, end;

    double time_spent;

    _initialize();

    begin = clock();

    printf("Fibonacci number is %d \n", fib(n));

    end = clock();

    time_spent = (double)(end - begin) / CLOCKS_PER_SEC;

    printf("\nTime Taken %lf", time_spent);

    return 0;
}

```

## Code for graph

```

import matplotlib.pyplot as plt

#nvalue=[40,39,41]

svalue=[39,40,41]

usingrec=[0.425173,0.639040,1.063056]

usingtabu=[0.000127,0.000138 ,0.000195]

plt.plot(svalue,usingrec)

plt.plot(svalue,usingtabu)

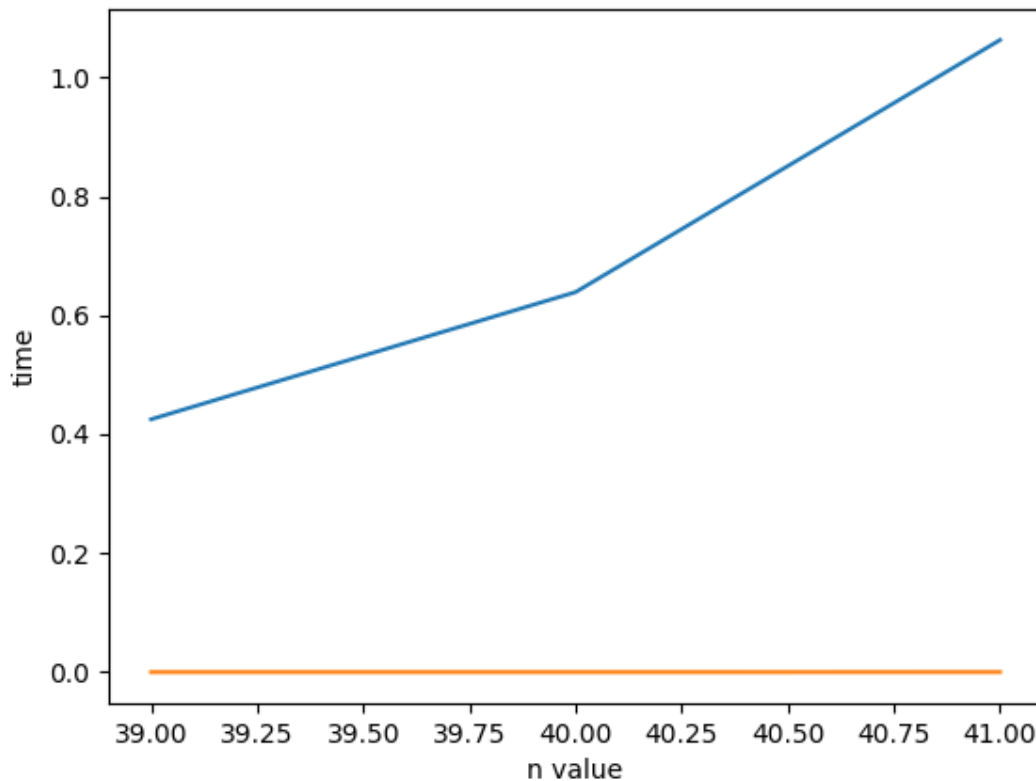
plt.xlabel("n value")

plt.ylabel("time")

```

```
# plt.plot(sam,sath)

plt.show()
```



Above curve is using recursion, below curve is using dynamic programming. We can observe how dynamic programming has decreased time complexity from exponential to polynomial.

## Longest common subsequence

### Problem Statement:

We have two sequences, we need to find the length of the longest subsequence present in both of them. A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous.

Ex: abcd is a sequence, then abc, acd, abd are subsequences

In order to find out the complexity of naive approach, we need to first know the number of possible different subsequences of a string with length  $n$ , i.e., find the number of subsequences with lengths ranging from  $1, 2, \dots, n-1$ . Recall from theory of permutation and combination that number of combinations with 1 element are  ${}^nC_1$ . Number of combinations with 2 elements are  ${}^nC_2$  and so forth and so on. We know that  ${}^nC_0 + {}^nC_1 + {}^nC_2 + \dots + {}^nC_n = 2^n$ . So a string of length  $n$

has  $2^n - 1$  different possible subsequences since we do not consider the subsequence with length 0. This implies that the time complexity of the brute force approach will be  $O(n * 2^n)$ . Note that it takes  $O(n)$  time to check if a subsequence is common to both the strings. This time complexity can be improved using dynamic programming.

### Optimal substructure property in this problem:

Let the input sequences be  $X[0..m-1]$  and  $Y[0..n-1]$  of lengths  $m$  and  $n$  respectively. And let  $L(X[0..m-1], Y[0..n-1])$  be the length of LCS of the two sequences  $X$  and  $Y$ . Following is the recursive definition of  $L(X[0..m-1], Y[0..n-1])$ .

If last characters of both sequences match (or  $X[m-1] == Y[n-1]$ ) then

$$L(X[0..m-1], Y[0..n-1]) = 1 + L(X[0..m-2], Y[0..n-2])$$

If last characters of both sequences do not match (or  $X[m-1] != Y[n-1]$ ) then

$$L(X[0..m-1], Y[0..n-1]) = \text{MAX} ( L(X[0..m-2], Y[0..n-1]), L(X[0..m-1], Y[0..n-2]) )$$

So the LCS problem has optimal substructure property as the main problem can be solved using solutions to subproblems.

### Overlapping subproblems property in this problem:

Suppose we have 2 sequences ABCF ,ACBD

$$\begin{array}{ccc} & \text{lcs("ABCF", "ACBD")} & \\ & / \quad \backslash & \\ \text{lcs("ABC", "ACBD")} & & \text{lcs("ABCF", "ACB")} \\ / & & / \end{array}$$

$\text{lcs("AB", "ACBD")}$   $\text{lcs("ABC", "ACB")}$   $\text{lcs("ABC", "ACB")}$   $\text{lcs("ABCF", "ACB")}$

In the above  $\text{lcs(ABC,ACB)}$  is computed twice.If we draw the complete recursion tree then we can see that there are many subproblems which are solved again and again. So this problem has Overlapping Substructure property and recomputation of same subproblems can be avoided by either using Memoization or Tabulation

### Code using normal recursion:

```
#include <stdio.h>
#include <time.h>
#include <string.h>
int max(int a, int b);
int lcs( char *X, char *Y, int m, int n )
{
    if (m == 0 || n == 0)
        return 0;
```

```

    if (X[m-1] == Y[n-1])
        return 1 + lcs(X, Y, m-1, n-1);
    else
        return max(lcs(X, Y, m, n-1), lcs(X, Y, m-1, n));
}
int max(int a, int b)
{
    return (a > b)? a : b;
}
int main()
{
    char X[] = "sathvikajjhggnsnkkeuh";
    char Y[] = "satikadnhdyuhdkkhhh";

    int m = strlen(X);
    int n = strlen(Y);
    clock_t begin, end;
    double time_spent;
    begin = clock();
    printf(" Length of lcs %d", lcs(X,Y,m,n));
    end = clock();
    time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
    printf("\nTime Taken %lf ", time_spent);
    return 0;
}

```

As we know that in recursion, for every recursive call a recursive function is splitting into two parts so we have a time complexity of  $O(2^n)$ .

### Code using dynamic programming:

```

#include <stdio.h>
#include <time.h>
#include <string.h>
int max(int a, int b);

int lcs(char *X, char *Y, int m, int n)
{
    int L[m + 1][n + 1];
    int i, j;
    for (i = 0; i <= m; i++)
    {
        for (j = 0; j <= n; j++)
        {
            if (i == 0 || j == 0)

```

```

        L[i][j] = 0;

        else if (X[i - 1] == Y[j - 1])
            L[i][j] = L[i - 1][j - 1] + 1;

        else
            L[i][j] = max(L[i - 1][j], L[i][j - 1]);
    }
}

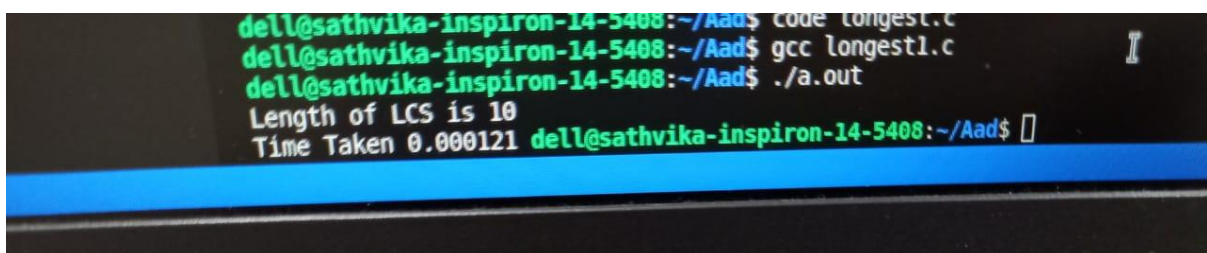
return L[m][n];
}

int max(int a, int b)
{
    return (a > b) ? a : b;
}

int main()
{
    char X[] = "sathvikajjhgggnkkeuh";
    char Y[] = "satikadnhdyuhdkkhhh";

    int m = strlen(X);
    int n = strlen(Y);
    clock_t begin, end;
    double time_spent;
    begin = clock();
    printf("Length of LCS is %d", lcs(X, Y, m, n));
    end = clock();
    time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
    printf("\nTime Taken %lf ", time_spent);
    return 0;
}

```



```

dell@sathvika-inspiron-14-5408:~/Aad$ gcc longestl.c
dell@sathvika-inspiron-14-5408:~/Aad$ ./a.out
Length of LCS is 10
Time Taken 0.000121 dell@sathvika-inspiron-14-5408:~/Aad$

```

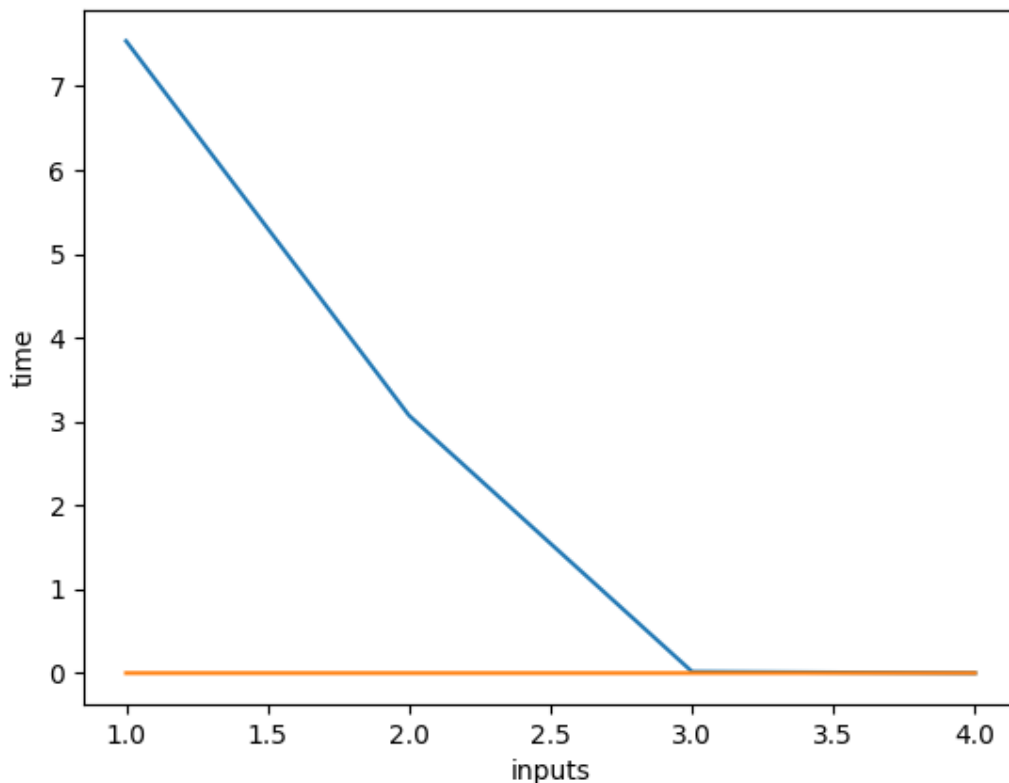
By storing every subproblem output and reusing them decreased the time complexity to  $O(mn)$ . because we are iterating  $m \times n$  times. We can observe using dynamic programming for the same inputs in recursion we have done in



7.539644 s but using dynamic programming it is 0.000121. we can observe huge variance in 2 methods.

#### Graph for comparing Time Complexities:

Inputs	Sequence1	Sequence2	Recursion	Dynamic
1	sathvikajjhg gsnkkeuh	satikadnhdyu hdkkhhh	7.539644	0.000121
2	sathvikajjhg gsnk	satikadnhdyu hdkk	0.167410	0.000129
3	sathvikajjhg gsnkpqr	satikadnhdyu hdkk	3.072352	0.000128
4	sathvika	samhitha	0.000133	0.000122



We can observe how dynamic programming runs fast for larger sequences.

## Word Wrap problem

Given a sequence of words, and a limit on the number of characters that can be put in one line (line width) Put line breaks in the given sequence such that the lines are printed neatly. Assume that the length of each word is smaller than the line width. The word processors like MS Word do the task of placing line breaks. The idea is to have balanced lines. In other words, not have a few lines with lots of extra spaces and some lines with a small amount of extra spaces. The extra spaces include spaces put at the end of every line except the last one. The problem is to minimize the following total cost.

Cost of a line = (Number of extra spaces in the line)<sup>3</sup>

Total Cost = Sum of costs for all lines

For example, consider the following string and line width  $M = 15$ . "mokey and pinky presents word wrap problem". Following is the optimized arrangement of words in 3 lines

mokey for pinky

presents word

wrap problem

Please note that the total cost function is not the sum of extra spaces, but the sum of cubes (or square is also used) of extra spaces. The idea behind this cost function is to balance the spaces among lines. For example, consider the following two arrangement of same set of words:

1) There are 3 lines. One line has 3 extra spaces and all other lines have 0 extra spaces. Total extra spaces =  $3 + 0 + 0 = 3$ . Total cost =  $3^3 + 0^3 + 0^3 = 27$ .

2) There are 3 lines. Each of the 3 lines has one extra space. Total extra spaces =  $1 + 1 + 1 = 3$ . Total cost =  $1^3 + 1^3 + 1^3 = 3$ .

Total extra spaces are 3 in both scenarios, but a second arrangement should be preferred because extra spaces are balanced in all three lines. The cost function with cubic sum serves the purpose because the value of total cost in the second scenario is less.

### Method 1 (Greedy Solution)

The greedy solution is to place as many words as possible in the first line. Then do the same thing for the second line and so on until all words are placed. This solution gives an optimal solution for many cases, but doesn't give an optimal solution in all cases. For example, consider the following string "aaa bb cc dddd" and line width as 6. Greedy method will produce the following output.

aaa bb  
cc  
dddd

Extra spaces in the above 3 lines are 0, 4 and 1 respectively. So the total cost is  $0 + 64 + 1 = 65$ .

But the above solution is not the best solution. Following arrangement has more balanced spaces. Therefore less value of total cost function.

aaa  
bb cc  
dddd

Extra spaces in the above 3 lines are 3, 1 and 1 respectively. So total cost is  $27 + 1 + 1 = 29$ .

Despite being suboptimal in some cases, the greedy approach is used by many word processors like MS Word and OpenOffice.org Writer.

### Method 2 (Dynamic Programming)

The following Dynamic approach strictly follows the algorithm given in the solution of Cormen book. First we compute costs of all possible lines in a 2D table  $lc[i][j]$ . The value  $lc[i][j]$  indicates the cost to put words from  $i$  to  $j$  in a single line where  $i$  and  $j$  are indexes of words in the input sequences. If a sequence of words from  $i$  to  $j$  cannot fit in a single line, then  $lc[i][j]$  is considered infinite (to avoid it from being a part of the solution). Once we have the  $lc[i][j]$  table constructed, we can calculate total cost using the following recursive formula. In the following formula,  $C[j]$  is the optimized total cost for arranging words from 1 to  $j$ .

The above recursion has overlapping subproblems. For example, the solution of subproblem  $c(2)$  is used by  $c(3)$ ,  $C(4)$  and so on. So Dynamic Programming is used to store the results of subproblems. The array  $c[]$  can be computed from left to right, since each value depends only on earlier values.

To print the output, we keep track of what words go on what lines, we can keep a parallel  $p$  array that points to where each  $c$  value came from. The last line starts at word  $p[n]$  and goes through word  $n$ . The previous line starts at word  $p[p[n]]$  and goes through word  $p[n] - 1$ , etc. The function `printSolution()` uses  $p[]$  to print the solution.

In the below program, input is an array  $l[]$  that represents lengths of words in a sequence. The value  $l[i]$  indicates the length of the  $i$ th word ( $i$  starts from 1) in the input sequence.

```
#include <limits.h>
#include <stdio.h>
#define INF INT_MAX

// A utility function to print the solution
int printSolution (int p[], int n);

// l[] represents lengths of different words in input sequence.
// For example, l[] = {3, 2, 2, 5} is for a sentence like
// "aaa bb cc dddd". n is size of l[] and M is line width
// (maximum no. of characters that can fit in a line)
void solveWordWrap (int l[], int n, int M)
{
    // For simplicity, 1 extra space is used in all below arrays
```

```

// extras[i][j] will have number of extra spaces if words from i
// to j are put in a single line
int extras[n+1][n+1];

// lc[i][j] will have cost of a line which has words from
// i to j
int lc[n+1][n+1];

// c[i] will have total cost of optimal arrangement of words
// from 1 to i
int c[n+1];

// p[] is used to print the solution.
int p[n+1];

int i, j;

// calculate extra spaces in a single line. The value extra[i][j]
// indicates extra spaces if words from word number i to j are
// placed in a single line
for (i = 1; i <= n; i++)
{
    extras[i][i] = M - l[i-1];
    for (j = i+1; j <= n; j++)
        extras[i][j] = extras[i][j-1] - l[j-1] - 1;
}

// Calculate line cost corresponding to the above calculated extra
// spaces. The value lc[i][j] indicates cost of putting words from
// word number i to j in a single line
for (i = 1; i <= n; i++)
{
    for (j = i; j <= n; j++)
    {
        if (extras[i][j] < 0)
            lc[i][j] = INF;
        else if (j == n && extras[i][j] >= 0)
            lc[i][j] = 0;
        else
            lc[i][j] = extras[i][j]*extras[i][j];
    }
}

```

```

// Calculate minimum cost and find minimum cost arrangement.
// The value c[j] indicates optimized cost to arrange words
// from word number 1 to j.
c[0] = 0;
for (j = 1; j <= n; j++)
{
    c[j] = INF;
    for (i = 1; i <= j; i++)
    {
        if (c[i-1] != INF && lc[i][j] != INF &&
            (c[i-1] + lc[i][j] < c[j]))
        {
            c[j] = c[i-1] + lc[i][j];
            p[j] = i;
        }
    }
}

printSolution(p, n);
}

int printSolution (int p[], int n)
{
    int k;
    if (p[n] == 1)
        k = 1;
    else
        k = printSolution (p, p[n]-1) + 1;
    printf ("Line number %d: From word no. %d to %d \n", k, p[n], n);
    return k;
}

// Driver program to test above functions
int main()
{
    int l[] = {5, 1, 5, 5};
    int n = sizeof(l)/sizeof(l[0]);
    int M = 6;
    solveWordWrap (l, n, M);
    return 0;
}

```

**Output:**

```
Line number 3: From word no. 4 to 4  
dell@sathvika-inspiron-14-5408:~/Aad$ gcc word.c  
dell@sathvika-inspiron-14-5408:~/Aad$ ./a.out  
Line number 1: From word no. 1 to 1  
Line number 2: From word no. 2 to 2  
Line number 3: From word no. 3 to 3  
Line number 4: From word no. 4 to 4  
dell@sathvika-inspiron-14-5408:~/Aad$
```