**MY DSA DOCUMENT**

**1.ALGORITHMIC TOOLBOX(COURSERA)-COPY**

**1.BASIC ALGORITHMS-(copy)**

**(I)Fibonacci Series**

**(ii)Greatest Common Divisors**

**(iii)Computing runtime**

**2.GREEDY ALGORITHM-(copy)**

**(i)Knapsack(fractional)**

**3.DIVIDE & CONCQUER ALGORITHM-(copy)**

**(i)Linear Search**

**(ii)Binary Search**

**(iii)Multiplying Polynimial**

**(iv)Master Theorem**

**(v)Sorting(Selection, Mergesort , Countingsort ,Quicksort)**

**1.Quicksort**

**Basically select a pivot element(last or first),elements less than pivot at left and greater elements at right.pivot in centre.then recursively quicksort left array and the right array.**

[**https://www.geeksforgeeks.org/quick-sort/**](https://www.geeksforgeeks.org/quick-sort/)

**4.DYNAMIC PROGRAMMING**

**(i)Coin Problem(copy)**

**(ii)Defination:**

* Dynamic is optimisation over plain recursion
* Simply store the results of subproblems so that we donot have to recompute them when needed later
* Reduces time complexity by saving time from computing same result again and again
* Eg-Fibonacci Series

Recursive Sol:

Int(fib int n)

{

If(n<1)

{

Return n;

}

Return fibo(n-1) + fib(n-2);

}

Dp Sol:

F[0]=0;

F[1]=1;

For(int i=2;i<n;i++)

{

F[i]=f[i-1]+F[i-2];

}

Return F[n];

* Main 2 type of DP:

(i)DP1-Overlapping Subproblems

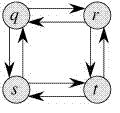
* + - For eg Fibonacci series
    - Combines solutions of subproblems
    - Solution-

(a)Memorization(Top Bottom)

(b)Tabuation(Bottom Top)

**(ii)**DP2-Optimal Structure

* + - A given problem have optimal structure if optimal sol of givn problem can be obtained by using optimal sol of its subproblem
    - For eg-Shortest Path



(a)Floyd Warshall

(b)Bellman -Ford

**(iii)Steps to solve DP problem:**

1) Identify if it is a DP problem

2) Decide a state expression with least parameters

3) Formulate state relationship

4) Do tabulation (or add memoization)

**Step 1 : How to classify a problem as a Dynamic Programming Problem ?**

* Most of the DP problems are to maximize or minimize
* All DP satisfy overlapping and c]some classic satify optimal substructure

**Step 2 : Deciding the state**

* State-Set of parameters uniquily identifying certain position in given problems
* State is the initial and imp step.
* The next state will be sub of initial state

**Step 3:Forming relation between states**

* Eg.

{1,3,5} given.we need to find total no. of ways possible to form given no. n;

For ex n=6

Total number of ways to form 6 is: 8

1+1+1+1+1+1

1+1+1+3

1+1+3+1

1+3+1+1

3+1+1+1

3+3

1+5

5+1

Thinking dynamically,first deciding state

State(n) coz n is the only parameter

State(1)=1;

State(2)=state(1)+1;

State(3)=state(2)+1 ||state(1)+2(cant add 2)

State(6)=state(5)+1 || state(3)+3 || state(1)+5;

::thus we can say state(n)= state(n-1)+1 || state(n-3)+1 || state(n-5)+1

Total number of ways **state(n) = state(n-1) + state(n-3) + state(n-5)**

**Code:**

// Returns the number of arrangements to

// form 'n'

int solve(int n)

{

// base case

if (n < 1)

return 0;

if (n == 1)

return 1;

return solve(n-1) + solve(n-3) + solve(n-5);

}

**Step 4: adding these states to the memory**

* we do this so that we don’t need to calculate each state again and again as the recurisive above is doing

code:

// initialize to -1

int dp[MAXN];

// this function returns the number of

// arrangements to form 'n'

int solve(int n)

{

  // base case

  if (n < 1)

      return 0;

  if (n == 0)

      return 1;

  if (n == 1)

      return 1;

  // checking if already calculated

  if (dp[n]!=-1)

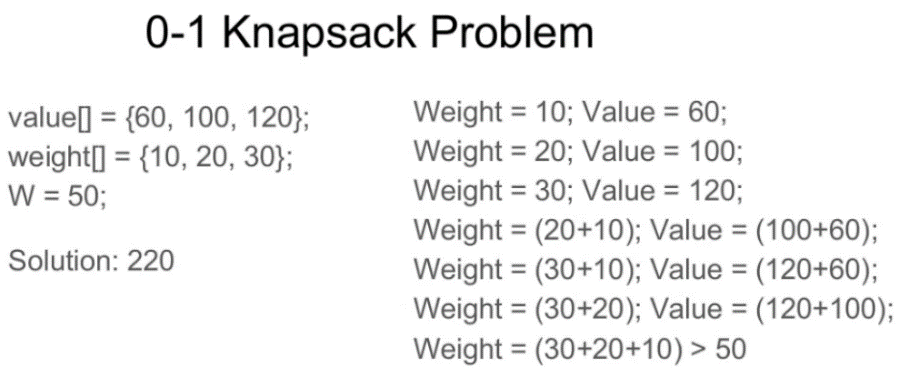
      return dp[n];

  // storing the result and returning

  return dp[n] = solve(n-1) + solve(n-3) + solve(n-5);

}

Knapsack Problem(non fractional)



1.Naive recursive

*/\* A Naive recursive implementation of   
 0-1 Knapsack problem \*/*#include **<bits/stdc++.h>  
using namespace** std;  
  
*// A utility function that returns   
// maximum of two integers***int** max(**int** a, **int** b) { **return** (a > b) ? a : b; }  
  
*// Returns the maximum value that   
// can be put in a knapsack of capacity W***int** knapSack(**int** W, **int** wt[], **int** val[], **int** n)  
{  
  
 *// Base Case* **if** (n == 0 || W == 0)  
 **return** 0;  
  
 *// If weight of the nth item is more   
 // than Knapsack capacity W, then   
 // this item cannot be included   
 // in the optimal solution* **if** (wt[n] > W)  
 **return** knapSack(W, wt, val, n - 1);  
  
 *// Return the maximum of two cases:   
 // (1) nth item included   
 // (2) not included* **else  
 return** max(  
 val[n] + knapSack(W - wt[n],  
 wt, val, n - 1),  
 knapSack(W, wt, val, n - 1));  
}  
  
*// Driver code***int** main()  
{  
 **int** val[] = { 60, 100, 120 };  
 **int** wt[] = { 10, 20, 30 };  
 **int** W = 50;  
 **int** n = **sizeof**(val) / **sizeof**(val[0]);  
 cout << knapSack(W, wt, val, n);  
 **return** 0;  
}

The recursion tree is for following sample inputs.

wt[] = {1, 1, 1}, W = 2, val[] = {10, 20, 30}

K(n, W)

K(3, 2)

/ \

/ \

K(2, 2) K(2, 1)

/ \ / \

/ \ / \

K(1, 2) K(1, 1) K(1, 1) K(1, 0)

/ \ / \ / \

/ \ / \ / \

K(0, 2) K(0, 1) K(0, 1) K(0, 0) K(0, 1) K(0, 0)

Recursion tree for Knapsack capacity 2

units and 3 items of 1 unit weight.

Time complexity=O(2^n)

Auxiliary Space=O(1)

2.DP

* In this method we are going to make 2D array instead of recursion.
* We basically take maximum between if the ith element will go into the bag or not if ith element weight is less than weight empty in bag thus comparing last row and current rows latest cell
* 2D array- column elements(ele 1,ele2 etc) column,max number of items bag can handle.

Let weight elements = {1, 2, 3}

Let weight values = {10, 15, 40}

Capacity=6

0 1 2 3 4 5 6

0 0 0 0 0 0 0 0

1 0 10 10 10 10 10 10

2 0 10 15 25 25 25 25

3 0

**Explanation:**

For filling 'weight = 2' we come

across 'j = 3' in which

we take maximum of

(10, 15 + DP[1][3-2]) = 25

| |

'2' '2 filled'

not filled

0 1 2 3 4 5 6

0 0 0 0 0 0 0 0

1 0 10 10 10 10 10 10

2 0 10 15 25 25 25 25

3 0 10 15 40 50 55 65

**Explanation:**

For filling 'weight=3',

we come across 'j=4' in which

we take maximum of (25, 40 + DP[2][4-3])

= 50

For filling 'weight=3'

we come across 'j=5' in which

we take maximum of (25, 40 + DP[2][5-3])

= 55

For filling 'weight=3'

we come across 'j=6' in which

we take maximum of (25, 40 + DP[2][6-3])

= 65

*// A Dynamic Programming based   
// solution for 0-1 Knapsack problem*#include **<stdio.h>***// A utility function that returns   
// maximum of two integers***int** max(**int** a, **int** b)  
{  
 **return** (a > b) ? a : b;  
}  
  
*// Returns the maximum value that   
// can be put in a knapsack of capacity W***int** knapSack(**int** W, **int** wt[], **int** val[], **int** n)  
{  
 **int** i, w;  
 **int** K[n + 1][W + 1];  
  
 *// Build table K[][] in bottom up manner* **for** (i = 0; i <= n; i++) {  
 **for** (w = 0; w <= W; w++) {  
 **if** (i == 0 || w == 0)  
 K[i][w] = 0;  
 **else if** (wt[i - 1] <= w)  
 K[i][w] = max(  
 val[i - 1] + K[i - 1][w - wt[i - 1]],  
 K[i - 1][w]);  
 **else** K[i][w] = K[i - 1][w];  
 }  
 }  
  
 **return** K[n][W];  
}  
  
**int** main()  
{  
 **int** val[] = { 60, 100, 120 };  
 **int** wt[] = { 10, 20, 30 };  
 **int** W = 50;  
 **int** n = **sizeof**(val) / **sizeof**(val[0]);  
 printf(**"%d"**, knapSack(W, wt, val, n));  
 **return** 0;  
}