**Greedy Algorithms:**

Greedy is an approach in which we select an item/activity which gives us the immediate benefit. That item/activity is then removed from the available list.

**Kruskal MST**: In Kruskal minimum spanning tree In Kruskal’s algorithm, we create a MST by picking edges one by one. The Greedy Choice is to pick the smallest weight edge that doesn’t cause a cycle in the MST constructed so far.

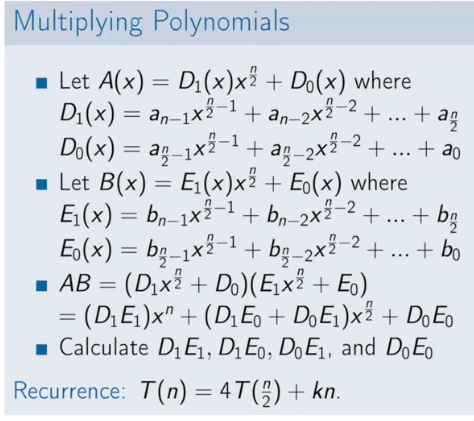
**Prim's MST and Dijkstra's Shortest path algorithm**: We maintain two sets: a set of the vertices already included in MST and the set of the vertices not yet included. The Greedy Choice is to pick the smallest weight edge that connects the two sets.

**Activity selection algorithm**: Sort the activities based on the end time and pick the first activity. After that go over the next set of activities which have start time greater than or = to finish time of current activity.

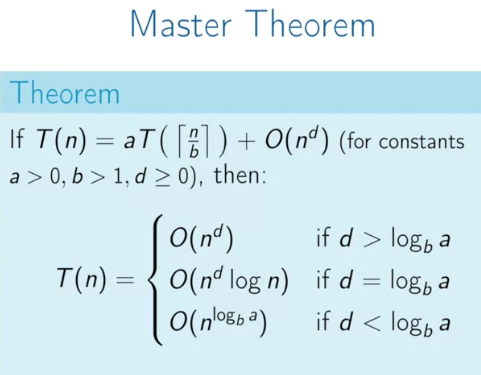
**Egyptian Fraction**: A unit fraction (which has den>num) can be further divided into multiple unit fractions. For this, find ceil of (den/num). Let it be ceil\_ then (1/ceil\_) will be one of the fraction. Recursively do for orig\_unit\_fraction - (1/ceil\_)

**Divide and Conquer**

Multiplying Polynomials

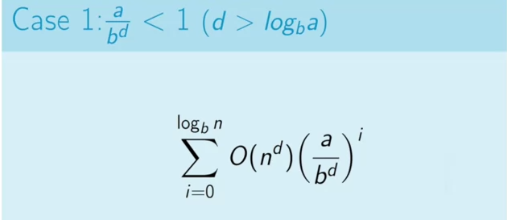


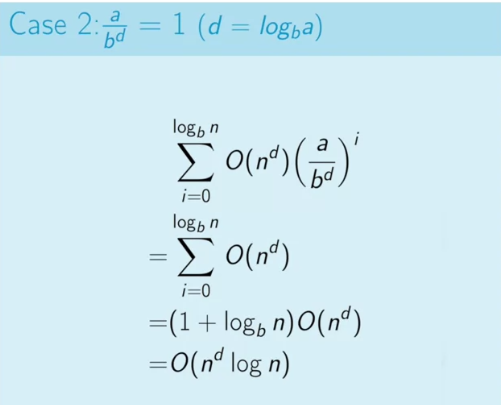
Master theorem to find the complexity of the recurrence relation mentioned as:

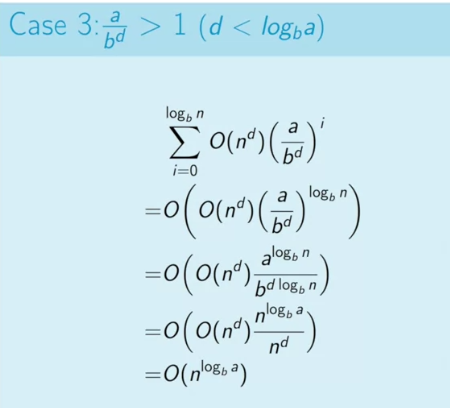


Everything relies on values of log(b)a and d. We have to choose max. So if d > log(b)a, keep it n^d, else if log(b)^a > n^d keep it log(b)^a else if both are equal, keep both as: n^d\*logn

Proof of master theorem:

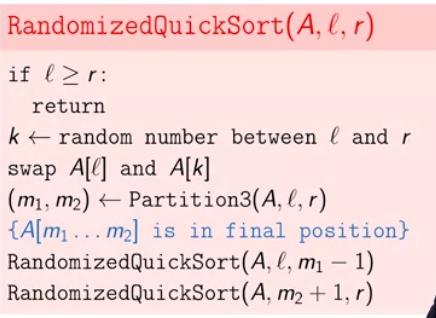






QuickSort

Quicksort has complexity of O(n^2), if all elements are equal.



Number of inversions:

An inversion of a sequence 𝑎0, 𝑎1, . . . , 𝑎𝑛−1 is a pair of indices 0 ≤ 𝑖 < 𝑗 < 𝑛 such

that 𝑎𝑖 > 𝑎*j.*

Use mergesort recursive calls and the following code:

/\* This function merges 2 arrays and count inversions of those \*/

int merge\_and\_count(vector<int>& vec, int left, int ave, int right) {

int tot\_sz = right-left+1;

int inversions = 0;

vector<int> result;

int i = left, j = ave+1;

while (i <= ave && j <= right) {

if (vec[i] <= vec[j]) { result.push\_back(vec[i]); ++i; }

else { inversions += (ave - i + 1); result.push\_back(vec[j]); ++j; }

}

if (i == ave+1) { for (int x = j; x <= right; ++x) result.push\_back(vec[x]); }}

else if (j == right+1) {for (int x = i; x < ave; ++x) result.push\_back(vec[x]);}}

// fill back

for (int i=left, j=0; i <= right && j<result.size(); ++i, ++j){vec[i] = result[j];}

return inversions;

}

**Covering the segments**

To capture the points which is covering the line segments, create a list of the form:

(a1l, l), (a2l, l), (p1, p), (a1r, r), (a3l, l), (a2r, r), (p2, p), (a3r, r)

sort it...

and then the Number of l's on the left side of point is the number of line segments it is covering.

**use this for solving problem on 1 axis**

**Dynamic Programming:**

DP is an approach for solving problems which seem recursive in nature but can be improvised in terms of time by building the solution from bottom to top rather than trying to solve the problem by solving the subproblems. DP problems exhibit 2 important properties:

>> **Overlapping Subproblems**: Like Divide and Conquer, Dynamic Programming combines solutions to sub-problems. Dynamic Programming is mainly used when solutions of 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. For example, Binary Search doesn’t have common subproblems.

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

DP problems can be solved using 2 approaches:

> **Memoization (Top-down):**

Memoization is a top-down approach in which we use a lookup table to check precomputed results of a subproblem. It is simply a modification of recursive solution such that we first check if solution to a given subproblem exists in table. If yes, retrieve value from table else compute it and store it back in table.

> **Tabulation (Bottom-up):**

Tabulation is a bottom-up approach which builds a table from bottom to top and finally returns the last element from the table. For example, for calculating Fibonacci number in bottom-up fashion, we first calculate fib(0) then fib(1) then fib(2) then fib(3) and so on. So literally, we are building the solutions of subproblems bottom-up.

**Notes:**

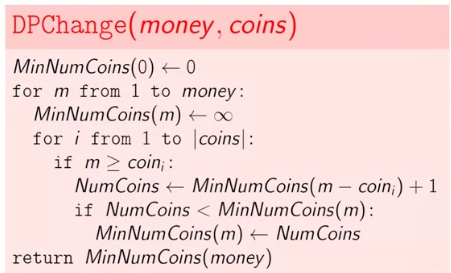
> All dynamic programming problems satisfy the overlapping subproblems property and most of the classic dynamic problems also satisfy the optimal substructure property. Once, we observe these properties in a given problem, be sure that it can be solved using DP.

> Typically, a greedy algorithm is used to solve a problem with optimal substructure if it can be proven by induction that this is optimal at each step. Otherwise, provided the problem exhibits overlapping subproblems as well, dynamic programming is used. If there are no appropriate greedy algorithms and the problem fails to exhibit overlapping subproblems, often a lengthy but straightforward search of the solution space is the best alternative.

**DP vs Greedy Approach:**

One of the example to quote when DP wins over greedy approach is coin change problem. Suppose we have coins of denominations - 20, 8 and 1. How many coins are required for changing 24. In greedy, we will first select 20 and then 4 single cent coins, giving a total of 5 coins, but we can also take 3 coins of 8 cents too :)

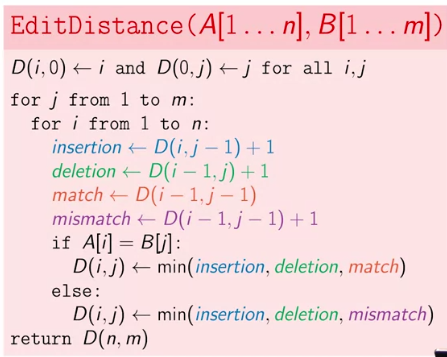
**Coin change problem**



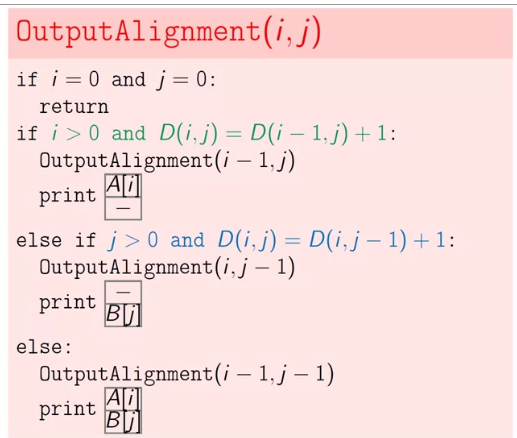
**Edit Distance Problem:**

In this problem, we are asked to find minimum number of insertions and deletion which need to be performed to make 1 string equal to another or in other words find the longest common subsequence.

Following is the algorithm to execute the necessary operations:



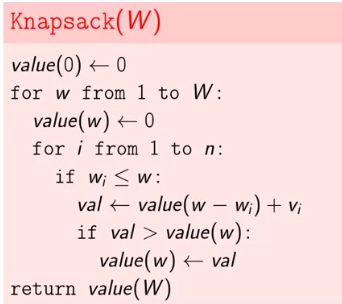
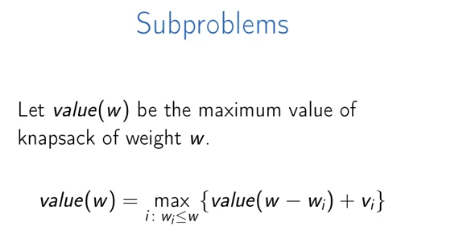
Following is the way we can backtrack and print the path:



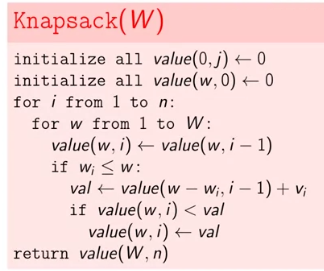
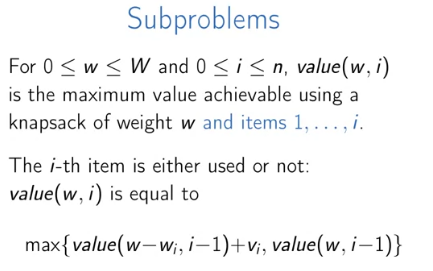
**Knapsack without repetition (1-D array used):**

Characterize the structure of optimal solution by creating optimal solution to a subproblem using cut and paste technique. Take out an item and its value

and declare that remaining weight should have optimal value if take out this item. After that fix that w(i) again in the total weight.



**Knapsack with repetition (2D-array used):**



For reconstructing the solution, or to actually calculate value compare the value of matrix[i-1][j] and matrix[i-1][j-weight[i-1]]

**Subset sum problem**:

> Set a[i][j] = a[i-1][j] if a[i-1][j] == 1. Since sum is already exist, else check for a[i-1][j-w[i]] given the fact that w[i] < j;

**Subset Sum problem with repetition:**

Recursion: count( S, m - 1, n ) + count( S, m, n-S[m-1] );

DP : 2 solutions

> Using 2D array:

Set dp[i][j] = dp[i][j-w[i]] else pick from top i.e

Dp[i][j] = dp[i-1][j]

> Using 1D array:

If(amount >= coin)

Combinations[amount] += Combinations[amount-coin]

Longest Common Subsequence

**Minimum number of insertions required to make a string palindrome:**

Length of string - (Length of LCS of string)

DP approach:

Let the input string be str[l……h]. The problem can be broken down into three parts:

1. Find the minimum number of insertions in the substring str[l+1,…….h].

2. Find the minimum number of insertions in the substring str[l…….h-1].

3. Find the minimum number of insertions in the substring str[l+1……h-1].

To Print all anagrams together

Sort those and store in hash if not exist else append in the form of hash chain.

To rotate an image by 90 degree get the transpose of the matrix and then reverse each row

Digraph processing is all about DFS

**Sorting**

Quicksort is better than HeapSort because there are no swaps made when the elements are in order.

nth order statistic or QuickSelect

function select(list, left, right, n)

if left = right // If the list contains only one element,

return list[left] // return that element

pivotIndex := ... // select a pivotIndex between left and right,

// e.g., left + floor(rand() \* (right - left + 1))

pivotIndex := partition(list, left, right, pivotIndex)

// The pivot is in its final sorted position

if n = pivotIndex

return list[n]

else if n < pivotIndex

return select(list, left, pivotIndex - 1, n)

else

return select(list, pivotIndex + 1, right, n)

**Trees**

**> The distance between two nodes can be obtained in terms of lowest common ancestor. Following is the formula.**

Dist(n1, n2) = Dist(root, n1) + Dist(root, n2) - 2\*Dist(root, lca)

> **To find if root to leaf path equals a certain sum:**

For root to leaf path sum do subsum = sum - node\_element if subsum is 0 in case left and right are NULL then that is the answer

**> To check if trees are similar :**

return ( areTreesSimilar(ptr1->left, ptr2->left) && areTreesSimilar(ptr1->right, ptr2->right) );

> **To check a sum tree:**

if(ptr->element != (isSumTree(ptr->left, var) + isSumTree(ptr->right, var)) )

=======================================================================

Distributed hashtables are those which are decentralized and for each key a value might exists on multiple nodes/clusters

DHTs only directly support exact-match search

To store just 100 numbers from incoming flow of numbers, use a circular buffer/heap can also be used

to sort a big array with many repetitions use AVL with repetitions

Anagrams

Sorting and counting the characters..we can use one array in which we increment one string and decrement for other. After operation, check if array is filled with 0s

To print 1 - 100

Either create 100 class objects or use recursive templates

To check if a string anagram can be palindrome

for this check if length is even then every character occurs even number of times

while if the length is odd, then one except one should appear even number of times

Fastest way to check 2 anagrams map each character with prime number ad find the multiplied result of both strings

if they are equal then it is or you can take a map / Character array[26] / Sorting

To find row wit hmax 1's first find the indes of 1 in first row and then update that index accordinly by going LHS and put max\_row\_index that row in case left entry

of original index is 1

to order one array based on another array

use a1 and put its values in hashmap with <ey, value> --> <number, occurences>

now traverse a2, search for that in hashmap if its found put that in p/p array and remove it from hashmap and rest of elements are sorted and appened to the o/p array

word break problem

if (dictionaryContains( str.substr(0, i) ) &&

wordBreak( str.substr(i, size-i) ))

return true;

#include<stdio.h>

/\* function to multiply two numbers x and y\*/

int multiply(int x, int y)

{

/\* 0 multiplied with anything gives 0 \*/

if(y == 0)

return 0;

/\* Add x one by one \*/

if(y > 0 )

return (x + multiply(x, y-1));

/\* the case where y is negative \*/

if(y < 0 )

return -multiply(x, -y);

}

Sum of subarray with a given sum divided by a number X... here in this suppose starting index is A and ending index is B then the remainder for the sum from 0->A and 0->b will be the same and between them will be the numbers with a given sum so for this

1, 9, 4, 5, 2, 8, 7, 11

sum array

1, 10 , 14, 19, 21, 29, 36, 47

now suppose X is 4 then mod array is

1 , 2, 2, 3, 1, 1, 0, 3

now create a hashmap with the <numbers\_given\_above, frequency>

total will be freq1 ! + freq2 ! + freq3 !... freqn !

here it will be - 0! + 3! + 1! + 1! = 6

**BitSets**

To find out number of set bits by

doing n = n&(n-1) 'x' times till n becomes 0 where x is number of bits set.

These method will fail if both variables are same so put a check before swapping

Get the rightmost set bit - set\_bit\_no = xor & ~(xor-1)

**Max profit**

fill sum array in reverse direction with each entry filled with the max element on its right

now traverse from left->right and get sum[i] - a[i] and add them to get max profilt

Align the most-significant ones of N and D.

* Compute t = (N - D);.
* If (t >= 0), then set the least significant bit of Q to 1, and set N = t.
* Left-shift N by 1.
* Left-shift Q by 1.
* Go to step 2.

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

{

b <<=1; // left shift b

b |= a & 0x1; //get unit bit

a >>= 1; // right shift a

}

Another trick to reverse a number

run loop from i = 0 to 16 (sizeof(data\_type) \* 4 )

check by right shifting number by & 1 | check by right shifting number by j & 1

set\_bit\_no = xor & ~(xor-1); ( rightmost set bit )

XOR of two different numbers x and y results in a number which contains set bits at the places where x and y differ. So if x and y are 10…0100 and 11…1001, then result would be 01…1101.

So the idea is to XOR all the elements in set. In the result xor, all repeating elements would nullify each other. The result would contain the set bits where two non-repeating elements differ.

Now, if we take any set bit of the result xor and again do XOR of the subset where that particular bit is set, we get the one non-repeating element. And for other non-repeating element we can take the subset where that particular bit is not set.

Find minimum using bitwise operators

y ^ ((x ^ y) & -(x < y))

Max

x ^ ((x ^ y) & -(x < y));

/\* This function will return n % d. d must be one of: 1, 2, 4, 8, 16, 32, … \*/

unsigned **int** getModulo(unsigned **int** n, unsigned **int** d)

{

**return** ( n & (d-1) );

}

2's complement of a number is 1's complement + 1

To add 1 to a number x (say 0011000111), we need to flip all the bits after the rightmost 0 bit (we get 001100**0**000). Finally, flip the rightmost 0 bit also (we get 0011001000) and we are done

To flip a number use XOR operator

/\* Flip all the set bits until we find a 0 \*/

while( x & m )

{

x = x^m;

m <<= 1;

}

check divisibility by 8

if (((x >> 3) << 3) == x) then done;

Boolean Array Puzzle

Given an array of 2 elements one is having value 0 and another one having value 1

make both 0.

Solution : a[ a[1] ] = a[ a[0] ]

=================================================================

**Arrays**

**Maximum bitonic subArray**

The bitonic array is one in which there are elements which are first in increasing order and then in decreasing order and you have to find maximum length of that array

Solution:

Construct two arrays int[] and dec[] in inc[] array make each element

inc[i]=inc[i-1]+1 if a[i] > a[i-1]

and then do the same in dec[] array but in opposite direction

i.e dec[i-1]=dec[i]+1 if a[i-1]>a[i]

At last find max ( inc[i] + dec[i] -1) for each i .

MAX Sub SEQUENCE SUM =

max[0] = A[0];

max[1] = A[1];

max[i] = max{ A[i-1], A[i-2] + A[i] }

To merge k sorted arrays use min heap putting all elements at once then picking min element

Snake and ladder game can be implemented using a 2-D array... to find min nodes use shortest path.

**Karumanchi**

k =1 -> n for logk = nlogn

1^p + 2^p + 3^p .... + n^p = ( n^(p+1))/(p+1)

Master Theorem for run time analysis

======================= Binary Search ======================

Matrix Search

Method 1: Find appropriate position of element in left column and right column. If the element is found in left/right column itself, return true, else check if index of 2 arrays are common. If not, return false else check for the array at that index.

Method 2: Set l = 0 and r = m\*n-1

to find mid = (l+r)/2

**bool** searchMatrix(vector**<**vector**<int>** **>** **&**matrix, **int** target) {

**int** n **=** matrix.size();

**int** m **=** matrix[0].size();

**int** l **=** 0, r **=** m **\*** n **-** 1;

**while** (l **!=** r){

**int** mid **=** (l **+** r **-** 1) **>>** 1;

**if** (matrix[mid **/** m][mid **%** m] **<** target)

l **=** mid **+** 1;

**else**

r **=** mid;

}

**return** matrix[r **/** m][r **%** m] **==** target;

}

**Strings**

**Knuth-Morris-Pratt**

Worst case complexity – O(n)

A string searching algorithm that runs in O(n) times as compare to naive algorithm which can has O(n\*m) as the worst complexity.

To accomplish this, we use the method of degeneration by calculating the length of the longest proper prefix in the (sub)pattern that matches a proper suffix in the same (sub)pattern.

e.g. In 'ABCAB', the prefix and suffix which are equal and are of maximum length is 'AB'

To carry out this task, we use an auxiliary array (also called partial match table) in which we store the maximum length of that prefix/suffix at a given position in pattern. e.g. for pattern 'ABCABLAB', the array will be

A|B|C|A|B|L|A|B|

0|0|0|1|2|0|1|2|

Let the string is 'ABCAPABCGCABCABLAB'

this table will help us deciding about how much characters we should skip in case of mismatch. e.g if pattern is 'ABCABLAB'

then we will have match for 'ABCA' (4 chars) and we will have mismatch for 5th character (required 'P' , got 'B') and in that case we will check the value at:

array[partial\_match\_length - 1] which is array[3] = 1 and we have to skip the search by

partial\_match\_length - array[partial\_match\_length - 1]

=4-1 = 2(we have to skip 'AB')