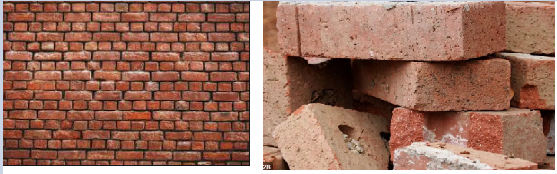
# Greedy Algorithm:

## What is greedy Algorithm?

Using the best solutions at the current level to build a solution. Algorithm that builds a program piece by Piece. It always choice the next best piece for immediate benefit.

Example: Building a Wall



## Why greedy algorithm?

When a local optimal solutions lead to fixing the global solution, the greedy algorithm works perfectly.

**Sorting**:

Insertion Sort

Selection sort

**Algorithm**:

Prim

Kruskal

# Known Algorithm:

**Insertion sort**

Divide into sorted and unsorted. Similar to selection, but here we pick the first element of unsorted array and put in the sorted array.

**3 1 5 2 6 4**

insertionsort(A)

loop i=1 to n-1

currentnumber=a[i], j=i-1

while a[j-1]>currentnumber && j>0

a[j]=a[j-1]

j++

a[j]= currentnumber

**Selection sort**

Finding min or max element in unsorted array and putting it in its correct position in a sorted array.

**3 1 5 2 6 4**

Put the result of min(find among all the element) in the first cell and make it as sorted. Then work on unsorted array for rest of the n-1 element.

selectionsort(A)

loop:j=0 to n-1

int min =j

loop i= j+1 to n-1

if (a[i]<a[min]

min =i-1

if min!=job

swap(a[j],a[min])

**Algorithm**:

**Prims**

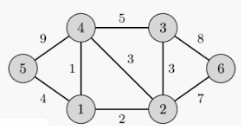
Take any vertex and mark its weight as 0 and rest as Infinite.

For every adjacent unvisited vertex, if current weight is more adjacent vertex edge, update the current weight with adjacent vertex’s edge.

Mark current vertex as visited.

Do above steps in increasing order of weight.

Print all the vertex with weight.

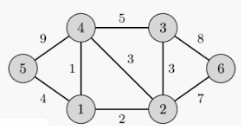


**Kruskal**

Finds min spanning tree for a connected weight graph.

By adding increasing cost arcs at each step.

Avoids cycle in every step.



# Activity Selection Problem

Select maximum number for cases from given N activity. Condition: One person can work in only one task at a time.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Activity | 1 | 2 | 3 | 4 |
| Start | 0 | 1 | 3 | 8 |
| Finish | 6 | 2 | 5 | 9 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Activity | 2 | 3 | 1 | 4 |
| Start | 1 | 3 | 0 | 8 |
| Finish | 2 | 5 | 6 | 9 |

**Algorithm**:

Sort the activities according to finish time 🡪 O (n log n)

Select first activity from the sorted array🡪 O (1)

If start time of next activity > the finish time of the current activity, ignore it. Else print it. 🡪 O (n)

**Algorithm**:

ActivitySelection(A)

Sort (A, finishTime)

previousActivity = FirstActivity

print FirstActivity

Loop i= 0 to n-1

If A[StartTime] of I > A [finishTime] of PreviousActivity

Print (A[StartTime],A[finishTime]

# Coin change problem:

**Problem**: We have (1, 2, 5, 10, 20, 50, 100, 500) currency notes. Find the minimum notes required to get the required value

Ex: 1600

Ex: 160

**Algorithm**:

Initialize value (V)

Find the largest denominator< V

Find the maximum value found by largest denomination and then subtract that from V

If v=0, print result

Else repeat 2 and 3 until condition gets satisfied.

**Program**:

CoinChange(N)

InitializeResult as empty

**Loop**: Until it breaks self

MaxDen = FindLargestDenominatorlessthanX

Insert (result, MaxDen)

N = N - MaxDen

If N = 0: Print result and exit

**Time complexity: 🡪 O(n) Space complexity 🡪 O(v)**

# Fractional Knapsack:

Problem: Fill the Knapsack with maximum value in a sack.

|  |  |  |
| --- | --- | --- |
| Item | Weight- Kg | Value |
| 1 | 6 | 9 |
| 2 | 10 | 5 |
| 3 | 8 | 8 |

Maximum weight allowed is 16 kg

**Algorithm**:

Calculate ratio of each Item

Sort Item based on ratio

Take item with highest ratio until weight allows

At the end add as much item as possible

**Program**:

Knapsack (capacity, W [][])

Ratio [] = calculateRatio (W)

SortDecending (ratio [])

Loop: i=0 to n-1

If(TotalW+ currentW<Capacity)

Consume(W[i][0])

TotalW= TotalW+ current

Else: ConsumeFraction (W[i][0]); Break;

**Time complexity: 🡪 O(n log n) Space complexity 🡪 O(v)**