# DESIGN AND ANALYSIS OF ALGORITHMS
# LAB 06-07: GREEDY ALGORITHMS

## I. Greedy Paradigm

The greedy algorithm makes a locally optimal solution. In some cases, this is enough to get a globally optimal solution.

- The choice at each step must be feasible: satisfy the requirements.
- Each step reduces the problem to a smaller problem.
- Locally optimal: Each step is Greedy and tries local optimization, that is, chooses best among local choices based on some metric.
- Short-sighted.
- Easy to construct, simple to implement
- Irrevocable: once choice is made, there is no going back.
- May or may not give optimal (best) solution.

## II. Visualization Example

```
import pylab
pylab.plot([1,2,3],[4,6,3],'o-')
```

## III. Runtime

```
import time
start_time = time.time()
main()
print("--- %s seconds ---" % (time.time() - start_time))
```

## IV. Input Generation

```
Import random
S = random.sample(range(100), 6)
print(S)
```

## V. Programming Exercises

1. Implement a greedy algorithm for the Coin Changing Problem in Python programming language and draw the running time of a program as a function of the size of its input, having generated different inputs.

   The Coin Changing Problem is described as follows

- Some coin denominations say, $1, 5, 10, 20, 50$
- Want to make change for amount $S$ using smallest number of coins.
- Example: Want change for $37$ cents.
  Optimal way is: $1 \times 20, 1 \times 10, 1 \times 5, 2 \times 1$.
- In some cases, there may be more than one optimal solution:
  Coins: $1, 5, 10, 20, 25, 50$,
  $S = 30$
  Then, $1 \times 20, 1 \times 10$ and $1 \times 25, 1 \times 5$ are both optimal solutions.
- Make a locally optimal choice: choose the largest coin possible at each step.

The greedy algorithm to solve The Coin Changing Problem is as follows

Greedy Solution:

Repeat
  Find the largest coin denomination (say $x$), which is $\leq S$.
  Use $\lfloor \frac{S}{x} \rfloor$ coins of denomination $x$.
  Let $S = S$ mod $x$.
until $S = 0$.

Input: Coin denominations $d[1] > d[2] > \ldots > d[m] = 1$.
Input: Value $S$
Output: Number of coins used for getting a total of $S$.
num$[i]$ denotes the number of coins of denomination $d[i]$.
  For $i = 1$ to $m$ do
    num$[i] = \lfloor \frac{S}{d[i]} \rfloor$.
    $S = S$ mod $d[i]$.
  EndFor

2. Implement a greedy algorithm for the Fractional Knapsack Problem in Python programming language and draw the running time of a program as a function of the size of its input, having generated different inputs.

   The Fractional Knapsack Problem is described as follows

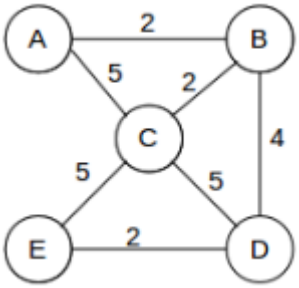
   - A knapsack of certain capacity (S Kg)
   - Various items available to carry in the knapsack.
   - There are $w_i$ Kg of item $i$ worth total of $c_i$.
   - You may pick fraction of some item $i$ (that is, you don't have to pick all $w_i$ Kg of item $i$). This will give prorated value for the item picked.

   The greedy algorithm to solve the Fractional Knapsack Problem is as follows

Suppose the items are $1, 2, \ldots, n$.
Weight of item $i$ is $w_i$ and value is $c_i$.
Rename the items such that they are sorted in
non-increasing order of $c_i/w_i$.
Thus, we assume below that $c_1/w_1 \geq c_2/w_2 \geq \ldots$.
$CapLeft = S$.
$TotalValuePicked = 0$.
For $i = 1$ to $n$ do
2. Pick $min(CapLeft, w_i)$ of item $i$.
3. $TotalValuePicked =$
$TotalValuePicked + (c_i/w_i) * min(CapLeft, w_i)$.
4. $CapLeft = CapLeft - min(CapLeft, w_i)$.
EndFor

Hint: For the following exercises related to graph, we can represent a edge as a tuple of three elements

```
>>> (1,5,4)
(1, 5, 4)
>>> x = (4,1,5)
>>> x[0]
4
>>> x[1]
1
>>> x[2]
5
>>> y = (2,3,4)
>>> z = [x,y]
>>> z
[(4, 1, 5), (2, 3, 4)]
>>> sorted(z)
[(2, 3, 4), (4, 1, 5)]
```



```
        G = {'A':[(2,'B'), (5, 'C')],'B':[(2,'A'),(2,'C'), (4,'D')]}
        >>> G['A']
        [(2, 'B'), (5, 'C')]
        >>> G['A'][0]
        (2, 'B')

        G = {'A':{'B':2, 'C':5},'B':{'A':2, 'C':2, 'D':4}}

        >>> G = {'A':{'B':2, 'C':5},'B':{'A':2, 'C':2, 'D':4}}
        >>> G['A']['C']
5
```

3. Implement a greedy algorithm for the Single Source Shortest Paths Problem in Python programming language and draw the running time of a program as a function of the size of its input, having generated different inputs.

The Single Source Shortest Paths Problem is described as follows

- Given a weighted graph, find the shortest path from a given vertex to all other vertices.
- Path (from $v_0$ to $v_k$): $v_0 v_1 \dots v_k$, such that $(v_i, v_{i+1})$ is an edge in the graph, for $i < k$.
- Simple Path: In the path, if $i \neq j$ then $v_i \neq v_j$.
- Weight of the path: sum of weights of the edges in the path.
- Shortest path: path with minimal weight.

- Intuition behind algorithm:
- Assuming positive integral distances between nodes.
- Walk in “all possible directions” for 1 step. If we encounter a node, then the shortest distance to it must be $1$.
  Then walk further for 1 step in all possible directions. If we encounter a node, not yet encountered, then the shortest distance to it must be $2$.
- Continue, similarly until all nodes are encountered.

- Graph $G = (V, E)$, with $wt(u, v)$ giving the weight of edge $(u, v)$
- Graph edges are given as adjacency list.
- $s$ is the source vertex.
- $Dist(u)$ denotes the currently best known distance from $s$ to $u$.
- Rem denotes the nodes for which the shortest distance is not yet found.
- $V - Rem$ will thus denote the vertices for which shortest distance is already found.
- $prev(v)$ denotes the predecessor node of $v$ in the currently known shortest path from $s$ to $v$.

The greedy algorithm (Dijkstra's Algorithm) to solve the Single Source Shortest Paths Problem is as follows

Set $Dist[u] = \infty$, and $Prev[u] = null$ for all $u \in V$.
Set $Dist[s] = 0$.
Set $Rem = V$.
While $Rem \neq \emptyset$ Do
1. Find node $u \in Rem$ with minimal $Dist[u]$.
2. $Rem = Rem - \{u\}$.
3. For each edge $(u, v)$ such that $v \in Rem - \{u\}$ Do
   3.1 Let $Z = min(Dist[v], Dist[u] + wt(u, v))$.
   3.2 If $Dist[v] > Z$, then let $Dist[v] = Z$ and $Prev(v) = u$
   Endif
   EndFor
End While

4. Implement Prim's Algorithm for finding a Minimal spanning tree in Python programming language and draw the running time of a program as a function of the size of its input, having generated different inputs.

The Minimal spanning tree Problem is described as follows

- Spanning tree of a graph $G = (V, E)$ is a tree $T = (V, E')$ such that $E' \subseteq E$.
  (Note: $G$ needs to be a connected graph)
- Minimal spanning tree of a weighted graph is a spanning tree of the graph with minimal weight.
- Note that there may be several minimal spanning trees.

The intuition to find MST:

- Start with one vertex (say start) to be in the spanning tree.
- Add an edge with minimum weight that has one endpoint in the tree already constructed, and the other endpoint outside the tree constructed. (Greedy!)
- Repeat until all vertices are in the tree
- Suppose $Rem$ denotes the set of vertices not yet in the spanning tree constructed.
- We will maintain an array $D(v)$ and $parent(v)$:
- For $v$ already in the tree, $parent(v)$ gives the node to which it is connected in the tree (at the time it was added to the tree). $parent(start) = null$.
- For $v$ in Rem, $D(v)$ gives the shortest distance from $v$ to the tree already constructed; $parent(v)$ gives the node in tree to which this shortest distance applies. Note that $D(v)$ may be $\infty$ in case there is no edge from $v$ to the nodes already in constructed tree. In this case $parent(v) = null$
- In each iteration, we will choose the vertex $u$ in $Rem$ which minimizes $D(u)$. We will add $u$ to the tree. Furthermore, we will update $D(v)$, for $v$ in $Rem - \{u\}$, in case, $wt(u, v) < D(v)$.

The Prim's Algorithm for finding a Minimal spanning tree is as follows

Prim's algorithm

1. Pick any node in $V$ as $start$.
2. Let $D(v) = \infty$ for all $v \in V - \{start\}$
3. Let $D(start) = 0$.
2. Let $parent(v) = null$ for all $v \in V$.
4. Let $Rem = V$.
5. While $Rem$ is not empty Do
   Choose a node $u$ in $Rem$ which minimizes $D(u)$.
   Delete $u$ from Rem.
   For each edge $(u, v)$ such that $v \in Rem - \{u\}$,
   If $D(v) > wt(u, v)$,
   then update $D(v) = wt(u, v)$ and
   $parent(v) = u$
   EndFor
   End While

5. Implement Kruskal's Algorithm for finding a Minimal spanning tree in Python programming language and draw the running time of a program as a function of the size of its input, having generated different inputs.

   The intuition of Kruskal's Algorithm:

- Edges are initially sorted by increasing weight.
- Start with $n$ isolated vertices (Forest)
- Grow the Forest by adding one edge at a time
- At each stage, add minimal weight edge which can be added without creating a cycle.
- Stop when $n-1$ edges have been added.

The Kruskal's Algorithm for finding a Minimal spanning tree is as follows

Kruskal; Input: $G = (V, E)$
Sort $E$ in non-decreasing order of the edge weights $w(e_1) \leq w(e_2) \ldots$.
$E_T \leftarrow \emptyset$; $counter \leftarrow 0$, $k \leftarrow 0$
While $counter < |V| - 1$ do
$k \leftarrow k + 1$.
If $E_T \cup \{e_k\}$ is acyclic
Then, $E_T \leftarrow E_T \cup \{e_k\}$; $counter \leftarrow counter + 1$.
Endwhile
End