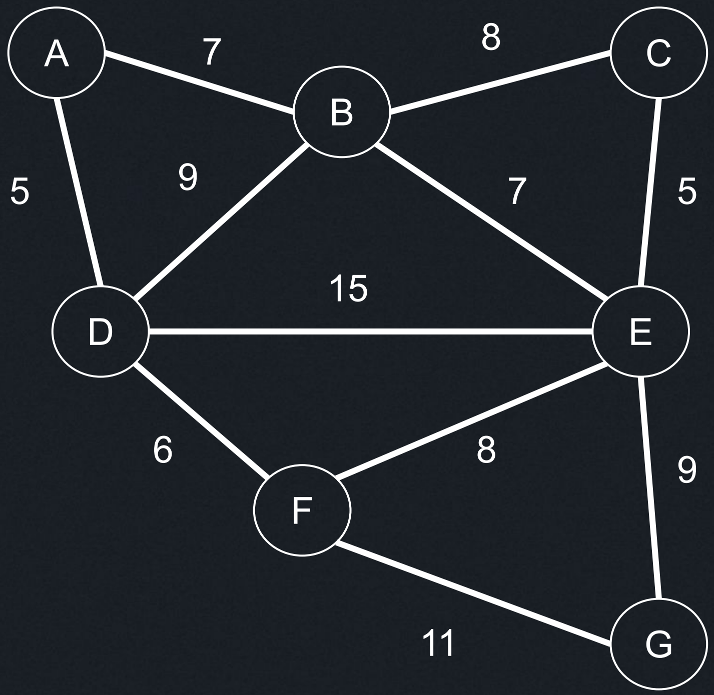
Graphs - Árvore Geradora Mínima

03/12/2024

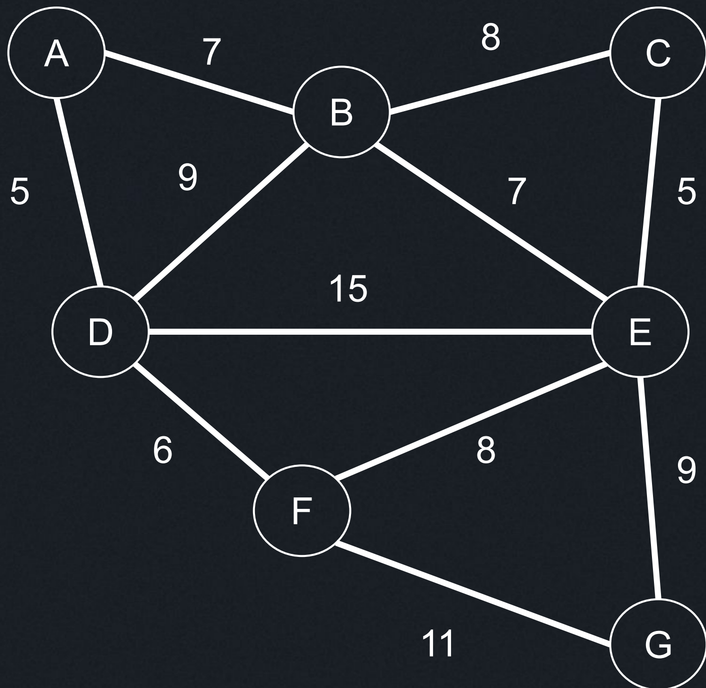
Árvore Geradora Mínima

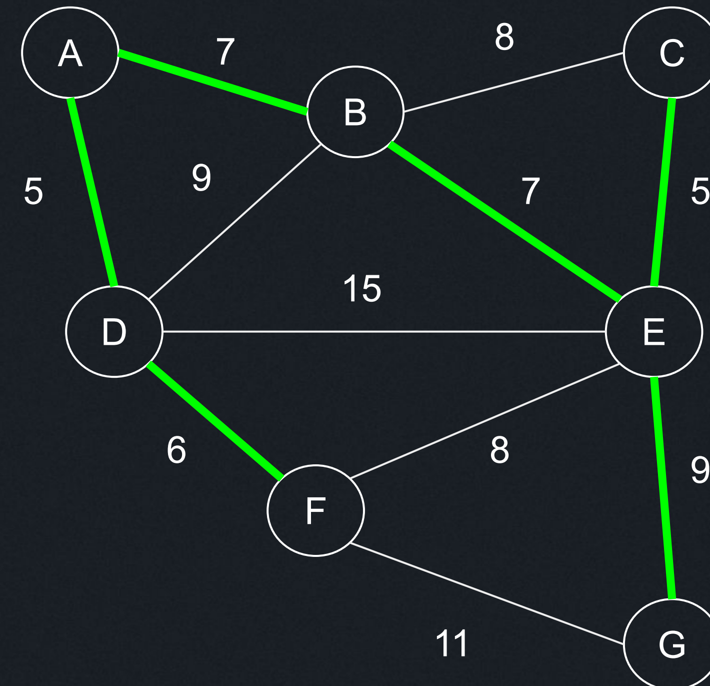


Árvore Geradora Mínima

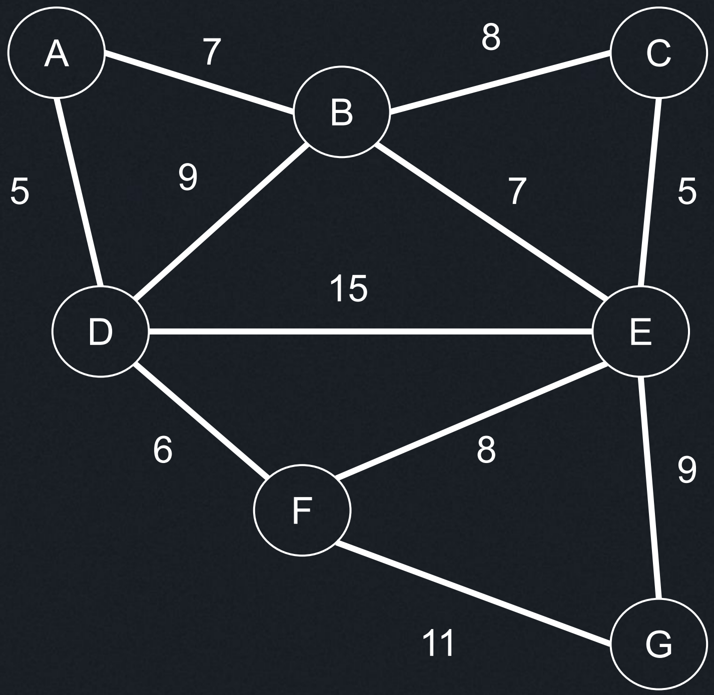
- Dado um grafo qualquer, encontrar uma árvore que conecta todos os vértices (nós) do grafo com soma total dos pesos das arestas mínima
- Dada a quantidade de nós V, são necessárias V - 1 arestas para manter a conectividade do grafo
- Pode ser útil em algoritmos de roteamento
  - garante que não existem ciclos
  - subconjunto de informações pode ser trocada entre roteadores / switches
- Outro exemplo, encontrar o custo mínimo de um projeto de cabeamento (excluindo as redundâncias que podem ser necessárias)
- Custo mínimo para conectar pontos em um espaço

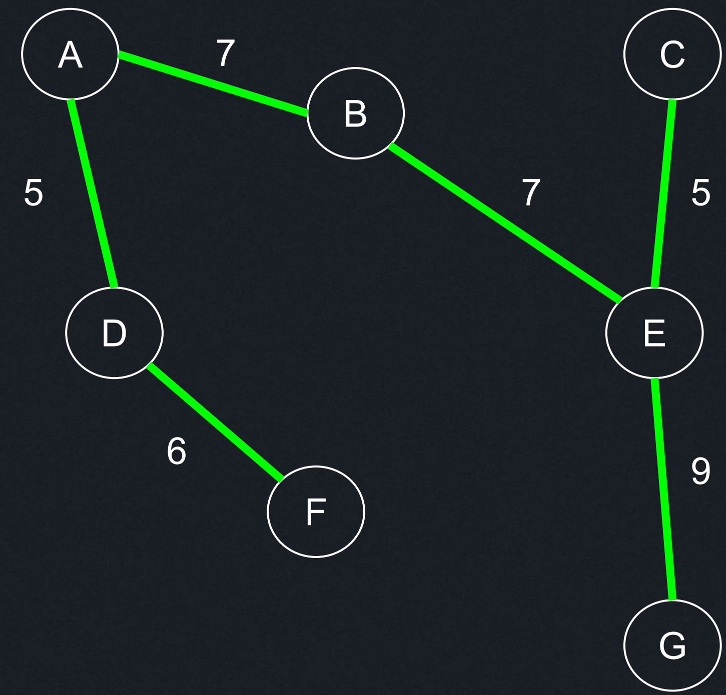
Árvore Geradora Mínima





Árvore Geradora Mínima

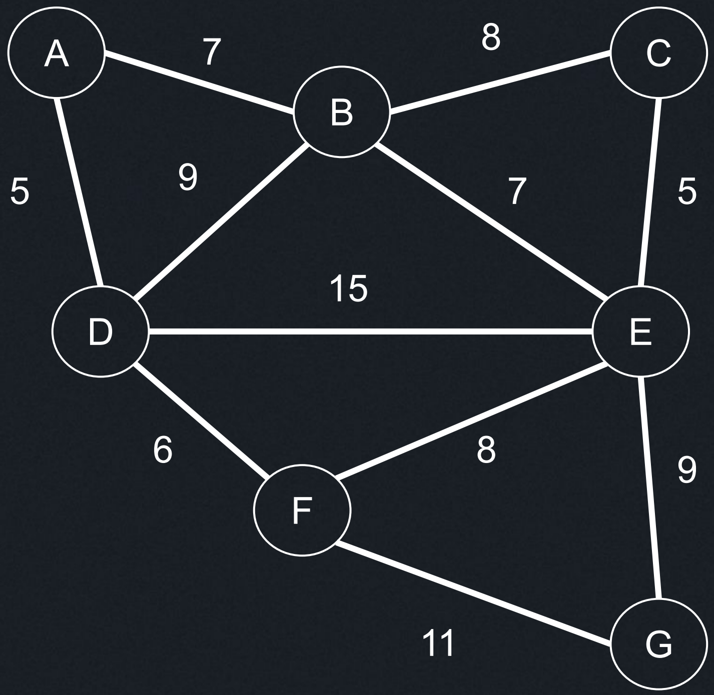




Árvore Geradora Mínima

- Dois algoritmos clássicos:
  - Algoritmo de Kruskal
    - melhor em grafos esparsos
  - Algoritmo de Prim
    - melhor em grafos densos

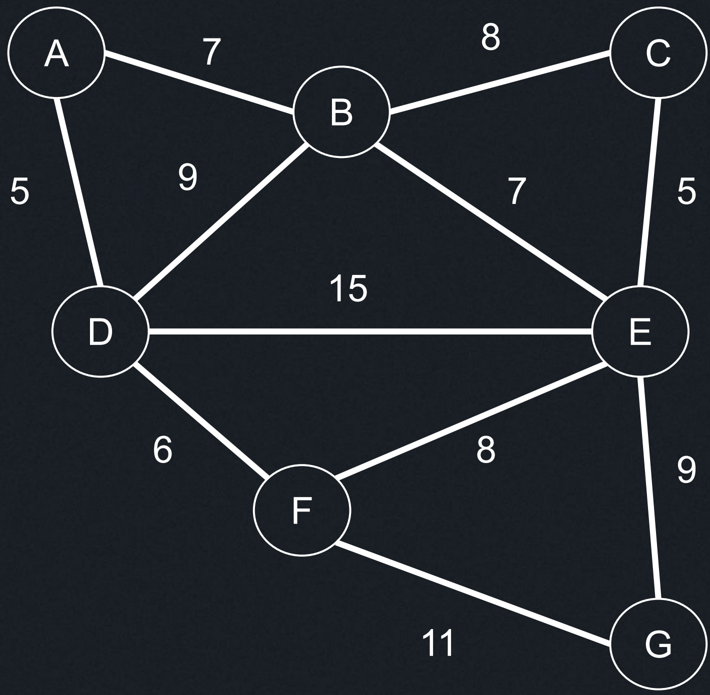
Algoritmo de Kruskal



5	A	D
5	C	E
6	D	F
7	A	B
7	B	E
8	B	C
8	E	F
9	B	D
9	E	G
11	F	G
15	D	E

- Passo 1: gerar um vetor com todas as arestas ordenadas pelos seus pesos

Algoritmo de Kruskal

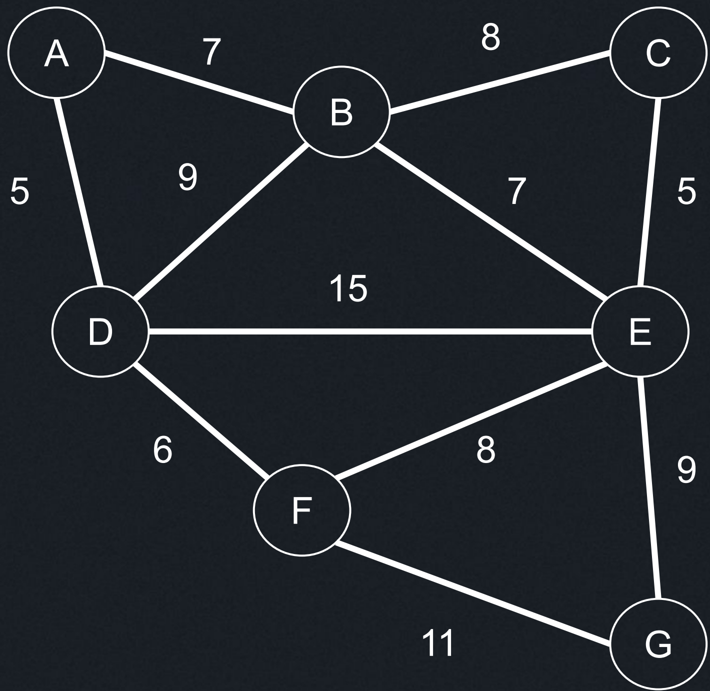


5	A	D
5	C	E
6	D	F
7	A	B
7	B	E
8	B	C
8	E	F
9	B	D
9	E	G
11	F	G
15	D	E

A	A
B	B
C	C
D	D
E	E
F	F
G	G

- Passo 2: criar subsets e colocar cada nó em um subset separado

Algoritmo de Kruskal

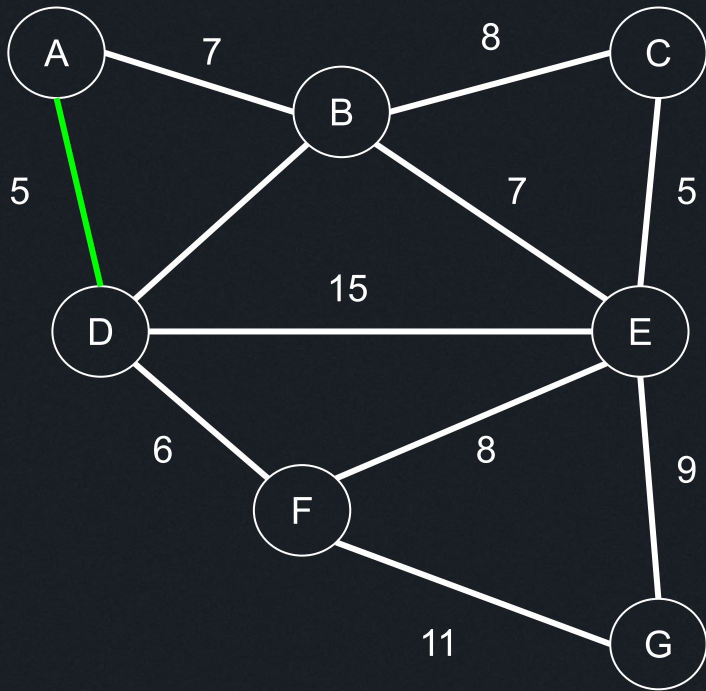


5	A	D
5	C	E
6	D	F
7	A	B
7	B	E
8	B	C
8	E	F
9	B	D
9	E	G
11	F	G
15	D	E

A	A
B	B
C	C
D	D
E	E
F	F
G	G

- Passo 3: iterar sobre as arestas, verificar se os nós estão no mesmo subset. Se não estiverem, colocá-los no mesmo subconjunto e incluir a aresta no conjunto solução

Algoritmo de Kruskal

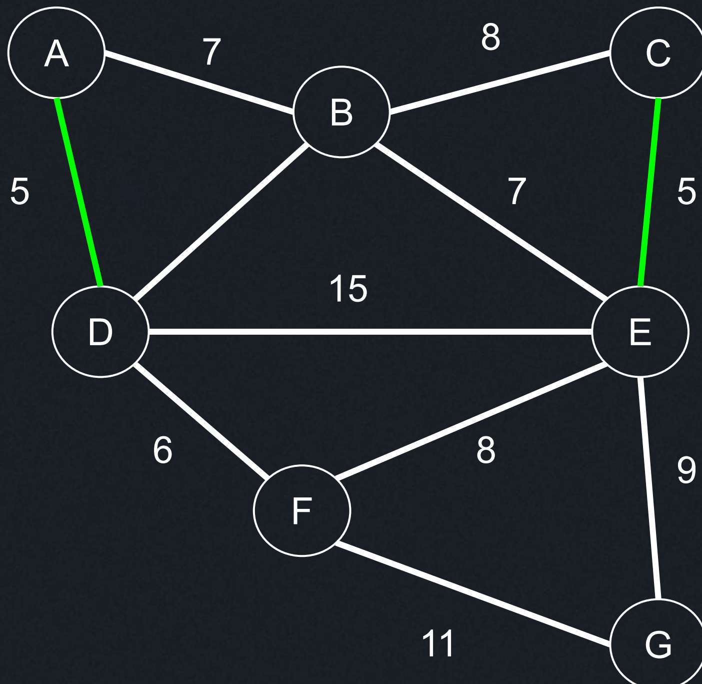


5	A	D
5	C	E
6	D	F
7	A	B
7	B	E
8	B	C
8	E	F
9	B	D
9	E	G
11	F	G
15	D	E

A	D
B	B
C	C
D	D
E	E
F	F
G	G

- Passo 3: iterar sobre as arestas, verificar se os nós estão no mesmo subset. Se não estiverem, colocá-los no mesmo subconjunto e incluir a aresta no conjunto solução

Algoritmo de Kruskal

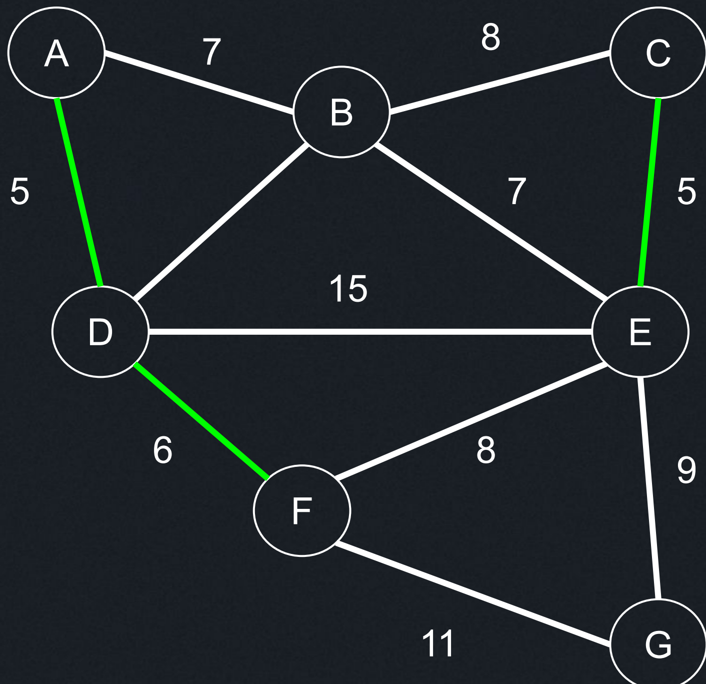


5	A	D
5	C	E
6	D	F
7	A	B
7	B	E
8	B	C
8	E	F
9	B	D
9	E	G
11	F	G
15	D	E

A	D
B	B
C	E
D	D
E	E
F	F
G	G

- Passo 3: iterar sobre as arestas, verificar se os nós estão no mesmo subset. Se não estiverem, colocá-los no mesmo subconjunto e incluir a aresta no conjunto solução

Algoritmo de Kruskal

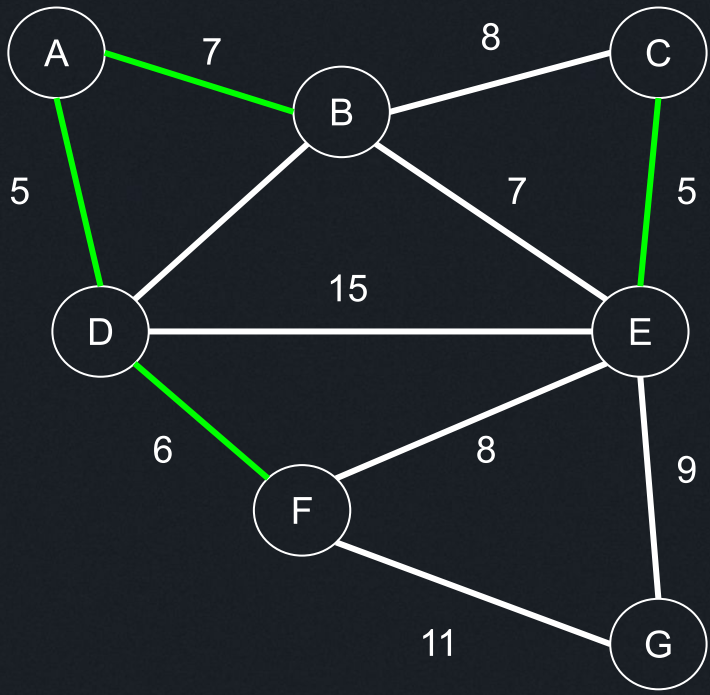


5	A	D
5	C	E
6	D	F
7	A	B
7	B	E
8	B	C
8	E	F
9	B	D
9	E	G
11	F	G
15	D	E

A	F
B	B
C	E
D	F
E	E
F	F
G	G

- Passo 3: iterar sobre as arestas, verificar se os nós estão no mesmo subset. Se não estiverem, colocá-los no mesmo subconjunto e incluir a aresta no conjunto solução

Algoritmo de Kruskal

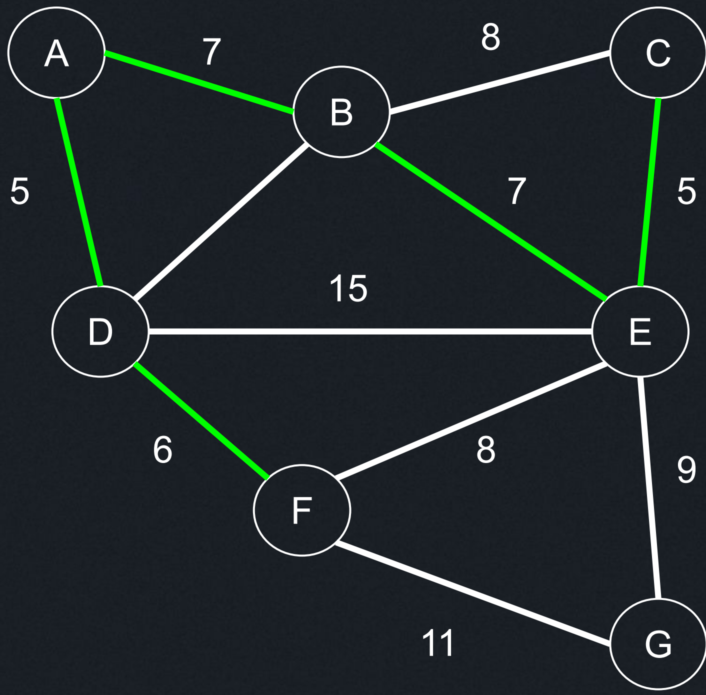


5	A	D
5	C	E
6	D	F
7	A	B
7	B	E
8	B	C
8	E	F
9	B	D
9	E	G
11	F	G
15	D	E

A	F
B	F
C	E
D	F
E	E
F	F
G	G

- Passo 3: iterar sobre as arestas, verificar se os nós estão no mesmo subset. Se não estiverem, colocá-los no mesmo subconjunto e incluir a aresta no conjunto solução

Algoritmo de Kruskal

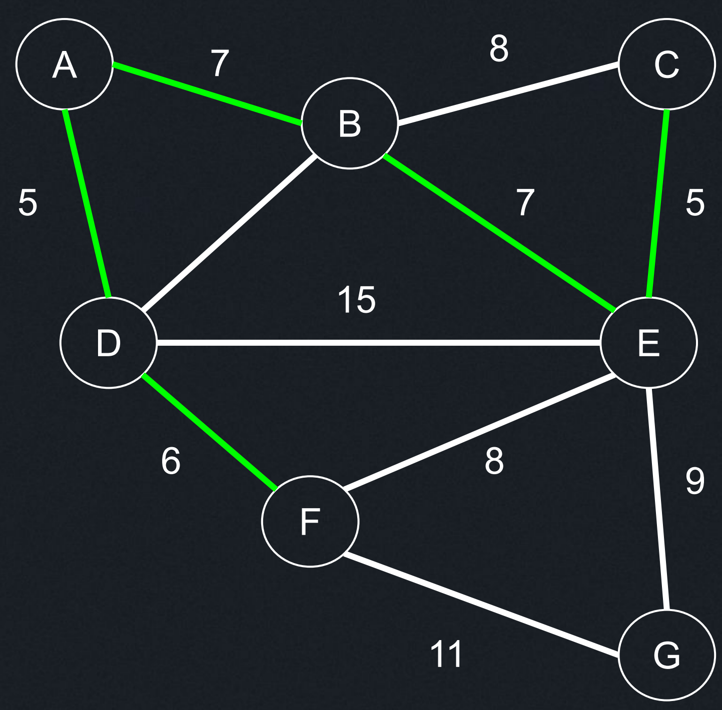


5	A	D
5	C	E
6	D	F
7	A	B
7	B	E
8	B	C
8	E	F
9	B	D
9	E	G
11	F	G
15	D	E

A	F
B	F
C	F
D	F
E	F
F	F
G	G

- Passo 3: iterar sobre as arestas, verificar se os nós estão no mesmo subset. Se não estiverem, colocá-los no mesmo subconjunto e incluir a aresta no conjunto solução

Algoritmo de Kruskal

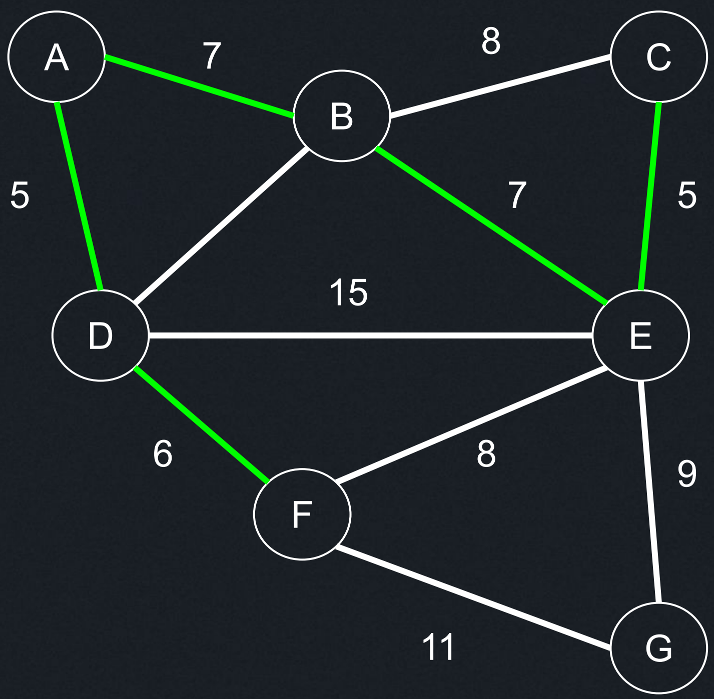


5	A	D
5	C	E
6	D	F
7	A	B
7	B	E
8	B	C
8	E	F
9	B	D
9	E	G
11	F	G
15	D	E

A	F
B	F
C	F
D	F
E	F
F	F
G	G

- Passo 3: iterar sobre as arestas, verificar se os nós estão no mesmo subset. Se não estiverem, colocá-los no mesmo subconjunto e incluir a aresta no conjunto solução

Algoritmo de Kruskal

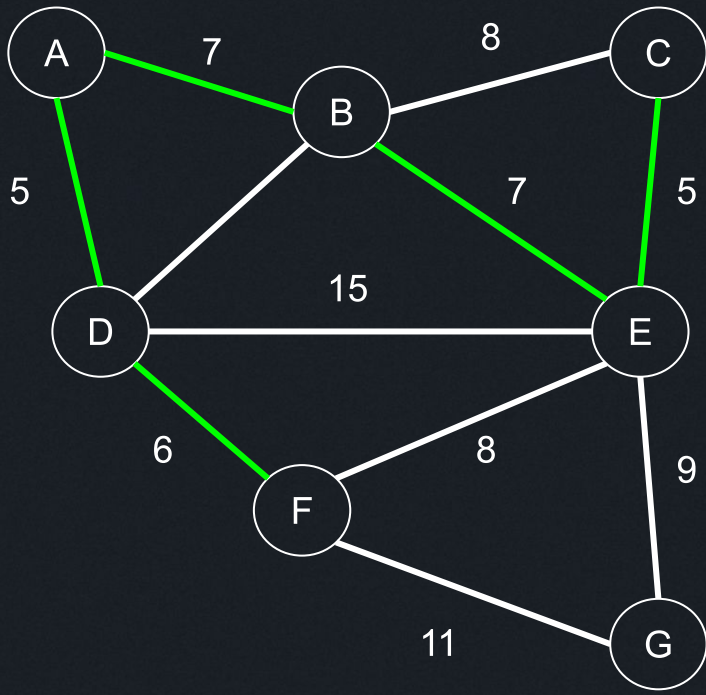


5	A	D
5	C	E
6	D	F
7	A	B
7	B	E
8	B	C
8	E	F
9	B	D
9	E	G
11	F	G
15	D	E

A	F
B	F
C	F
D	F
E	F
F	F
G	G

- Passo 3: iterar sobre as arestas, verificar se os nós estão no mesmo subset. Se não estiverem, colocá-los no mesmo subconjunto e incluir a aresta no conjunto solução

Algoritmo de Kruskal

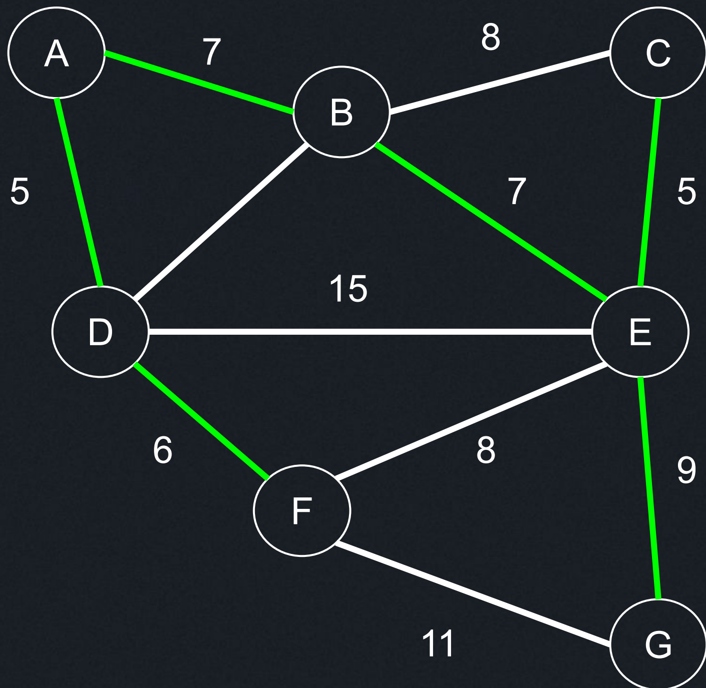


5	A	D
5	C	E
6	D	F
7	A	B
7	B	E
8	B	C
8	E	F
9	B	D
9	E	G
11	F	G
15	D	E

A	F
B	F
C	F
D	F
E	F
F	F
G	G

- Passo 3: iterar sobre as arestas, verificar se os nós estão no mesmo subset. Se não estiverem, colocá-los no mesmo subconjunto e incluir a aresta no conjunto solução

Algoritmo de Kruskal



5	A	D
5	C	E
6	D	F
7	A	B
7	B	E
8	B	C
8	E	F
9	B	D
9	E	G
11	F	G
15	D	E

A	F
B	F
C	F
D	F
E	F
F	F
G	F

- 5 + 5 + 6 + 7 + 7 + 9 = 39

Árvore Geradora Mínima

- Critério de parada: atingir V - 1 arestas

Algoritmo de Kruskal sem Conjuntos Disjuntos

```
def Kruskal_NoDisjointSets(self):
        # build array of edges
        edges = []
        for node in range(self.V): # O(E)
            for i in range(len(self.adj[node])):
                neigh = self.adj[node][i]
                weight = self.weight[node][i]
                edges.append( (weight, node, neigh) )

        edges.sort() # O(ElogE)

        # build subsets
        subsets = [i for i in range(self.V)] # O(V)
```

Algoritmo de Kruskal sem Conjuntos Disjuntos

```
def Kruskal_NoDisjointSets(self):
        # build array of edges and subsets # O(E(1 + log(E)) + O(V)
        ...

        sum_of_edges = 0
        count_edges = 0
        for weight, source, target in edges: # O(E)
            if subsets[source] != subsets[target]:
                sum_of_edges += weight
                save_subset = subsets[source]
                for i in range(self.V): # O(V)
                    if subsets[i] == save_subset:
                        subsets[i] = subsets[target]
                count_edges += 1
                if count_edges == self.V-1:
                    break

        return sum_of_edges
```

Algoritmo de Kruskal sem Conjuntos Disjuntos

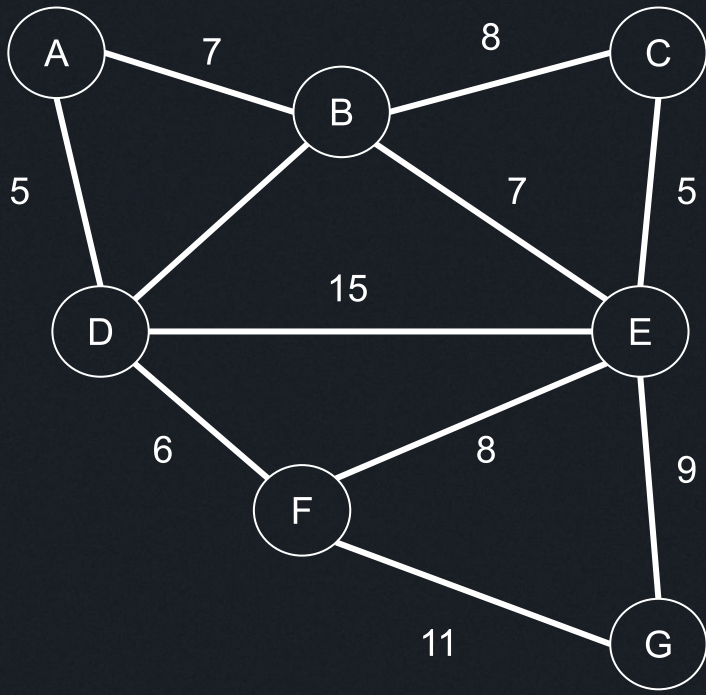
```
def Kruskal_NoDisjointSets(self):  # O(E + ElogE + V + EV) => O(E(1 + log(E)
+ V)) => O(ElogE + EV) => O(E(logE + V)) => V^2(2logV + V)
        # build array of edges and subsets  # O(E(1 + log(E) + V)
        ...

        sum_of_edges = 0
        count_edges = 0
        for weight, source, target in edges:  # O(E)
            if subsets[source] != subsets[target]:
                sum_of_edges += weight
                save_subset = subsets[source]
                for i in range(self.V):  # O(V)
                    if subsets[i] == save_subset:
                        subsets[i] = subsets[target]
                count_edges += 1
                if count_edges == self.V-1:
                    break

        return sum_of_edges
```

Conjuntos Disjuntos: FIND + UNION

- FIND:
  - recebe um elemento e identifica a qual subconjunto pertence um elemento

- UNION:
  - recebe dois elementos e faz a união dos subconjuntos

Algoritmo de Kruskal

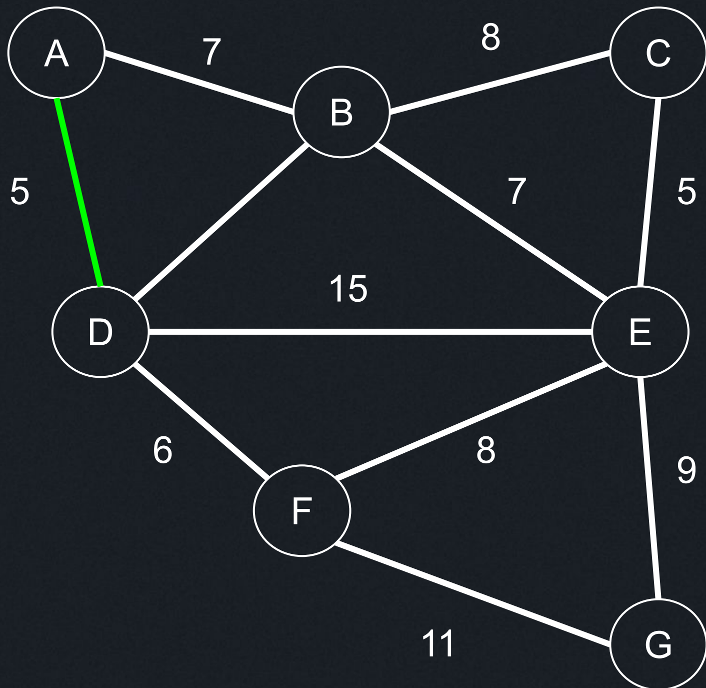


5	A	D
5	C	E
6	D	F
7	A	B
7	B	E
8	B	C
8	E	F
9	B	D
9	E	G
11	F	G
15	D	E

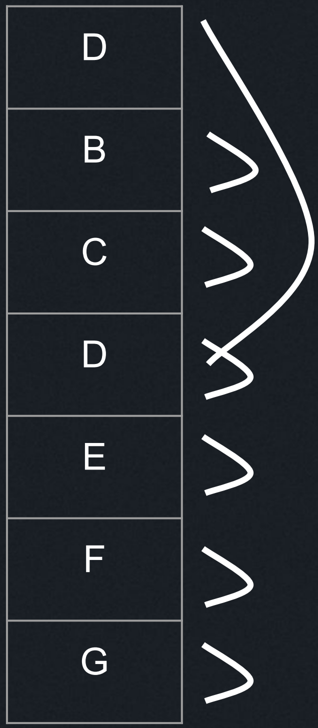
A
B
C
D
E
F
G

ada

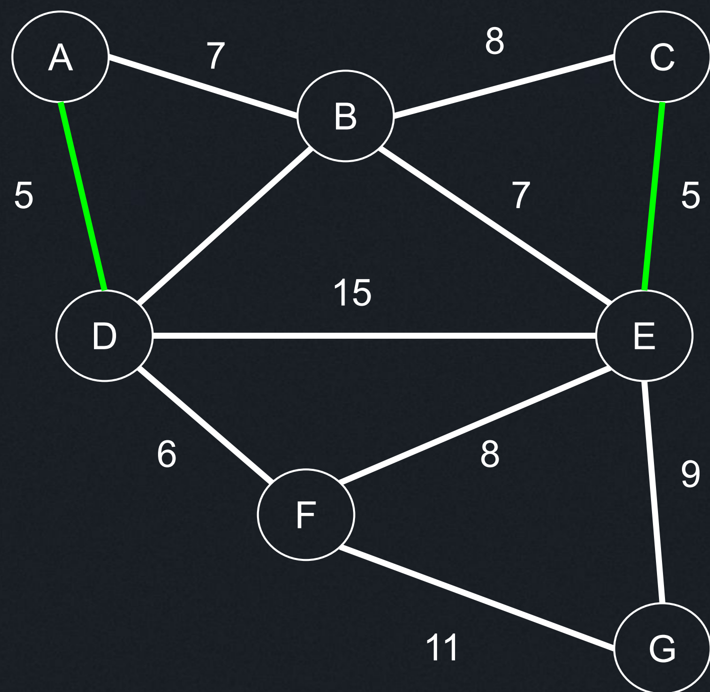
Algoritmo de Kruskal



5	A	D
5	C	E
6	D	F
7	A	B
7	B	E
8	B	C
8	E	F
9	B	D
9	E	G
11	F	G
15	D	E



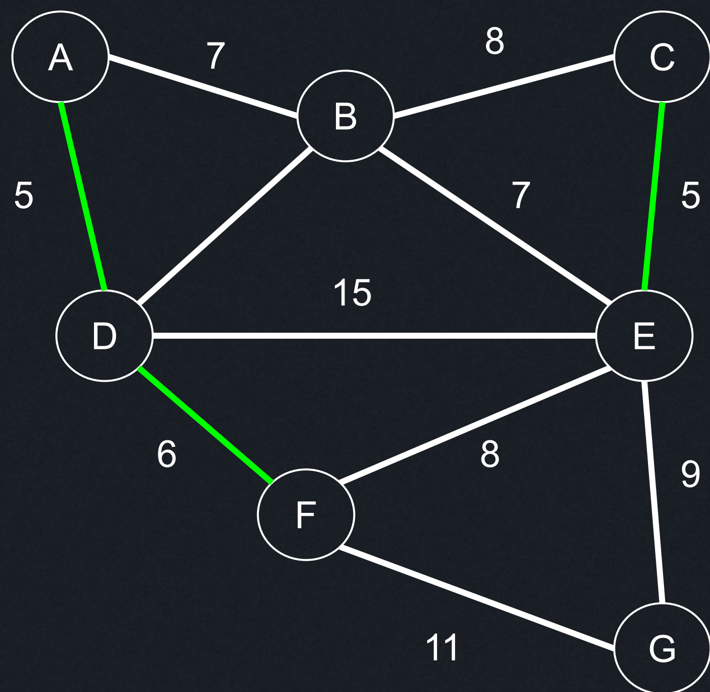
Algoritmo de Kruskal



5	A	D
5	C	E
6	D	F
7	A	B
7	B	E
8	B	C
8	E	F
9	B	D
9	E	G
11	F	G
15	D	E

D
B
E
D
E
F
G

Algoritmo de Kruskal



5	A	D
5	C	E
6	D	F
7	A	B
7	B	E
8	B	C
8	E	F
9	B	D
9	E	G
11	F	G
15	D	E

F
B
E
F
E
F
G

- União passa a ser: "ir na raiz do subconjunto e mudá-la para unir-se ao outro subconjunto

Conjuntos Disjuntos: FIND + UNION

- FIND:
  - recebe um elemento e identifica a qual subconjunto pertence um elemento
- UNION:
  - recebe dois elementos e faz a união dos subconjuntos

Conjuntos Disjuntos: FIND + UNION

```
class DisjointSets:
    def __init__(self, size):
        self.dj = [i for i in range(size)]

    def find(self, elem):
        if self.dj[elem]!=elem:
            self.dj[elem] = self.find(self.dj[elem])
        return self.dj[elem]

    def union(self, elem1, elem2):
        self.dj[ self.find(elem1) ] = self.find(elem2)
```

Conjuntos Disjuntos: FIND + UNION

```
class DisjointSets:
    def __init__(self, size):
        self.dj = [i for i in range(size)]

    def find(self, elem):  # log(size)
        if self.dj[elem]!=elem:
            self.dj[elem] = self.find(self.dj[elem])
        return self.dj[elem]

    def union(self, elem1, elem2):  # log(size)
        self.dj[ self.find(elem1) ] = self.find(elem2)
```

Algoritmo de Kruskal com Conjuntos Disjuntos

```
def Kruskal(self):
        # build array of edges
        ...

        subsets = DisjointSets(self.V)

        sum_of_edges = 0
        count_edges = 0
        for weight, source, target in edges:
            if subsets.find(source)!=subsets.find(target):
                sum_of_edges += weight
                subsets.union(source, target)
                count_edges += 1
                if count_edges==self.V-1:
                    break

        return sum_of_edges
```

Algoritmo de Kruskal com Conjuntos Disjuntos

```
def Kruskal(self):
        # build array of edges  # O(E(1 + log(E)) => O(Elog(E))
        ...

        subsets = DisjointSets(self.V)  # O(V)

        sum_of_edges = 0
        count_edges = 0
        for weight, source, target in edges:  # O(E)
            if subsets.find(source)!=subsets.find(target):  # log(V)
                sum_of_edges += weight
                subsets.union(source, target)  # log(V)
                count_edges += 1
                if count_edges==self.V-1:
                    break

        return sum_of_edges
```

Algoritmo de Kruskal com Conjuntos Disjuntos

```
def Kruskal(self): # O(Elog(E) + V + Elog(V)) => O(Elog(V) + V) => O(Elog(V))
        # build array of edges  # O(E(1 + log(E)) => O(Elog(E))
        ...

        subsets = DisjointSets(self.V)  # O(V)

        sum_of_edges = 0
        count_edges = 0
        for weight, source, target in edges:  # O(E)
            if subsets.find(source)!=subsets.find(target):  # log(V)
                sum_of_edges += weight
                subsets.union(source, target)  # log(V)
                count_edges += 1
                if count_edges==self.V-1:
                    break

        return sum_of_edges
```

Min Cost to Connect All Points

https://leetcode.com/problems/min-cost-to-connect-all-points/description/

1584. Min Cost to Connect All Points

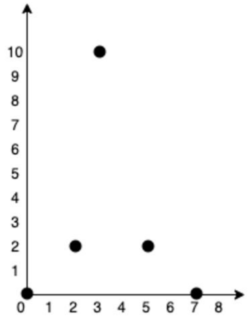
Medium | Topics | Companies | Hint

You are given an array `points` representing integer coordinates of some points on a 2D-plane, where $\texttt{points}[i] = [x_i, y_i]$.

The cost of connecting two points $[x_i, y_i]$ and $[x_j, y_j]$ is the **manhattan distance** between them: $|x_i - x_j| + |y_i - y_j|$, where $|val|$ denotes the absolute value of `val`.

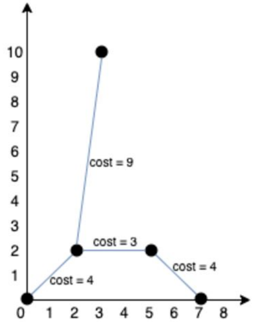
Return *the minimum cost to make all points connected.* All points are connected if there is **exactly one** simple path between any two points.

Example 1:



```
Input: points = [[0,0],[2,2],[3,10],[5,2],[7,0]]
Output: 20
```

```
Explanation:
```




```
We can connect the points as shown above to get the minimum cost of 20.
Notice that there is a unique path between every pair of points.
```

Min Cost to Connect All Points

https://leetcode.com/problems/min-cost-to-connect-all-points/description/

```
def manhattanDistance (self, p1, p2):
        return abs(p1[0] - p2[0]) + abs(p1[1] - p2[1])

def minCostConnectPoints(self):
        num_points = len(points)

        edges = []
        for i in range(num_points):
            for j in range(i + 1, num_points):
                weight = self.manhattanDistance(points[i], points[j])
                edges.append( (weight, i, j) )

        edges.sort()
        ...
```

ada

Min Cost to Connect All Points

https://leetcode.com/problems/min-cost-to-connect-all-points/description/

```
def minCostConnectPoints(self):
       ...
       subsets = DisjointSets(num_points)
       sum_of_edges = 0
       count_edges = 0
       for weight, source, target in edges:
           if subsets.find(source)!=subsets.find(target):  # log(V)
               sum_of_edges += weight
               subsets.union(source, target)  # log(V)
               count_edges += 1
               if count_edges==num_points:
                   break

       return sum_of_edges
```

ada

Interleaving String

https://leetcode.com/problems/interleaving-string/

97. Interleaving String

Medium | Topics | Companies

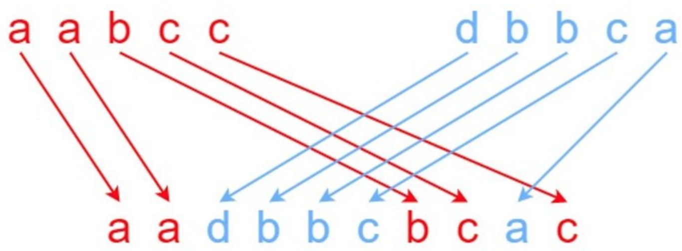
Given strings `s1`, `s2`, and `s3`, find whether `s3` is formed by an **interleaving** of `s1` and `s2`.

An **interleaving** of two strings `s` and `t` is a configuration where `s` and `t` are divided into `n` and `m` substrings respectively, such that:

- $s = s_1 + s_2 + ... + s_n$
- $t = t_1 + t_2 + ... + t_m$
- $|n - m| <= 1$
- The **interleaving** is $s_1 + t_1 + s_2 + t_2 + s_3 + t_3 + ...$ or $t_1 + s_1 + t_2 + s_2 + t_3 + s_3 + ...$

Note: `a + b` is the concatenation of strings `a` and `b`.

Example 1:




```
Input: s1 = "aabcc", s2 = "dbbca", s3 = "aadbbcbcac"
Output: true
Explanation: One way to obtain s3 is:
Split s1 into s1 = "aa" + "bc" + "c", and s2 into s2 = "dbbc" + "a".
Interleaving the two splits, we get "aa" + "dbbc" + "bc" + "a" + "c" = "aadbbcbcac".
Since s3 can be obtained by interleaving s1 and s2, we return true.
```

ada

Interleaving String

https://leetcode.com/problems/interleaving-string/

```
def isInterleave(self, s1, s2, s3):
        n1, n2, n3 = len(s1), len(s2), len(s3)
        if n3 != n1 + n2: return False

        attempted = set()
        def rec(i1, i2):
            if (i1, i2) in attempted: return False

            i3 = i1 + i2
            if i3 == n3: return True

            attempted.add((i1, i2))
            return (
                    i1 < n1 and s1[i1] == s3[i3] and rec(i1 + 1, i2)
                 or i2 < n2 and s2[i2] == s3[i3] and rec(i1, i2 + 1)
            )

        return rec(0, 0)
```

ada

Obrig.ada