Consider the following recurrence relation. Let $T_1, \cdots, T_n$ be a sequence of n positive integers. For a given n and c, we define $M(i,j)$, $0 \le i \le n$, $0 \le j \le c$ as follows

LPA

Exam No:

$$M(i,j) = \begin{cases} \text{true} & \text{if } j = 0 \text{ and } i \ge 0 \\ \text{false} & \text{if } j > 0 \text{ and } i = 0 \\ M(i-1,j) \vee M(i-1, j-T_i) & \text{if } j > 0, i > 0 \text{ and } j - T_i \ge 0 \\ M(i-1,j) & \text{if } j > 0, i > 0 \text{ and } j - T_i < 0 \end{cases}$$

where $\vee$ is the logical or.

Give the pseudo-code of a bottom-up dynamic programming algorithm that exploes the recurrence above to find the value for $M(n,c)$, for a given n & c. Discuss its time complexity.

DYNAMIC PROGRAMMING

- Solve an optimization problem by caching subproblem solutions, rather than recomputing them.

1) **Optimal substructure**
   An optimal solution to a problem contains optimal solutions to sub-problems
2) **Overlapping sub-problems**
   The solution to sub-problems can be reused several times

Our approach to a given problem

1) Find a suitable notion of subproblem → both properties hold (using induction)
2) Define the recurrence for that notion of subproblem
3) Build a recursive algorithm
4) Build a topdown D.P.A.
5) Build a bottomup D.P.A

Longest increasing subsequence

SUBPROBLEM Given a sequence $S = (S_1, \cdots, S_n)$, let $LIS(i)$ be the longest increasing subsequence that ends with S_i

The longest among $LIS(1), LIS(2), \cdots, LIS(n)$ gives the solution to the problem

OPTIMAL SUBSTRUCTURE PROPERTY

If s_i is removed from LIS(i), we obtain { LIS(j) ; $s_j < s_i$, $j < i$ / the empty sequence }

Proof Proof by contradiction

1) **ASSUMPTION** Assume that LIS(i) is the LIS that ends with s_i
2) **NEGATION** Now, assume that $|LIS(j)| > |LIS(i) \setminus \{s_i\}|$
3) **CONSEQUENCE** Then, appending s_i to LIS(j) generate a sequence longer than LIS(i): $|LIS(j) \cup \{s_i\}| > |LIS(i)|$
4) **CONTRADICTION** But this leads to a contradiction of 1

⇒ Therefore, $LIS(i) \setminus \{s_i\}$ must be ~~LIS(i)~~ LIS(j)

LIS can be solved recursively

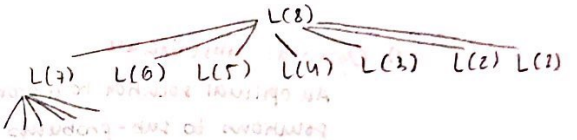
0, 8, 4, 12, 2, 10, 6, 14
(1) (2) (3) (4) (5) (6) (7) (8)

The size of the LIS is given by the maximum of $L_1, L_2, \ldots, L_n$

Function LIS(S, i)

BASE CASE
if i = 1 then
  $L_1 = 1$ se estamos no primeiro elemento, então o tamanho da LIS que acaba nesse é 1
else
  $L_i = \emptyset$
  for j = 1 to i-1 do
    L_j = LIS(S, j)
    if $s_j < s_i$ and $L_j > L_i$ then } Optimal substructure
      $L_i = L_j$
  $L_i = L_i + 1$ adds this element to the LIS that ends in s_i
return L_i

The recursion tree



⇓

L(i) can be cached

⇓

TOP-DOWN DP

Function LIS(S, i)

```
if LIS[i] is cached then
    return LIS[i]
if i = 1 then
    LIS[i] = 1
else { LIS[i] = ∅
    for j = 1 to i-1 do
        LIS[j] = LIS(S, j)
        if s_j < s_i and LIS[j] > LIS[i] then
            LIS[i] = LIS[j]
    LIS[i] = LIS[j] + 1.
return LIS[i]
```

There are # $O(n)$ overlapping sub-problems, which suggests a $O(n^2)$ dynamic programming algorithm **BOTTOM UP**

1. For each position $i = 1, \ldots, n$, find the largest LIS for positions $j < i$ such that $s_j < s_i$. Append s_i to it
2. Return the largest LIS found

```
Function LIS (S)
    LIS[1] = 1
    for i = 2 to n do
        LIS[i] = 0
        for j = 1 to i - 1 do
            if s_j < s_i and LIS[j] > LIS[i] then
                LIS[i] = LIS[j]
        LIS[i] = LIS[i] + 1.
    return max(LIS[1], ..., LIS[n])
```

escolhe os subproblemas ótimos para construir a solução de um problema maior

How to reconstruct an optimal subsequence?

Start from the largest LIS & scan from right to left, choosing a smaller number with next unitary decrement in # LIS

COIN CHANGING
Subset sum

Find the change for $C' \leq C$ with minimum number of coins using the first $i \leq n$ coin denominations

(1) Let S be the set with the minimum number of coins to change C', taken from the first i denominations, and using a coin denomination d_i
(2) Then S without that coin is optimal for $C' - d_i$

Optimal substructure

1) If the optimal solution for the problem above contains a coin with denomination i, then, by removing it, we have an optimal solution for the change without that coin.

2) If the optimal solution for the problem above does not contain a coin with denomination i, then, we have an optimal solution for the same change for the first $i - 1$ denominations

PROOF BY CONTRADICTION

Negate 2 ⇒ Assume that you can find a change for $C' - d_i$ with less coins using the first i denominations

contradict 1 ⇒ Then, it is also possible to change C' with less coins by adding a coin with denomination # d_i

Recursive Approach

For denomination i:
- use denomination d_i & make the change for $C' - d_i$ with the denominations available
- OR
- Do not use denominations d_i & make the change for C' with the remaining denominations

⇓

Choose the minimum of the two

THE KNAPSACK PROBLEM

Subproblem

Find the objects taken the first is n objects that maximize the value and satisfy the constraint $W' \leq W$

Let S be the optimal set of objects, taken from the first i objects, with total value v and total weight $w \leq W'$

Optimal Substructure

If S contains the i-th object, then, by removing it, we have an optimal solution with objects taken from the first $i-1$ objects that satisfies the constraint without the weight of that object.

If S does not contain the i-th object, then we have an optimal solution with objects taken from the first $i-1$ objects that satisfies constraint W'

(1) Let S be the optimal set of objects, taken from the first i objects, with total value v and weight $w \leq W'$, using the i-th object.

(2) Then, S without the i-th object, with total value $v - v_i$ & total weight $w - w_i$, is optimal for the first $i-1$ objects & satisfies constraint $W' - w_i$

PROOF BY CONTRADICTION

Negate (2) ⇒ Assume that there exists another set of objects taken from the first $i-1$ objects, with total value $v' > v - v_i$ and total weight $w' \leq W' - w_i$

Contradict (1) ⇒ Then, there also exists a set using the i-th object with total value $v' + v_i > v$ & weight $w' + w_i \leq W'$

Recursive solution → Choose the nth object

(1) Either use it & solve sub-problem for $w - w_n$ with the remaining $n-1$ objects

OR

(2) Do not use it & solve subproblem for W with the remaining $n-1$ objects

(3) Choose the max value of the two

$T_1, \ldots, T_n$ => sequence of n positive integers

$$M(i,j) = \begin{cases} \text{true} & \text{if } j \leq 0 \text{ and } i \geq 0 \\ \text{false} & \text{if } j > 0 \text{ and } i = 0 \end{cases} \Big\} \text{ base case}$$

$$\begin{cases} M(i-1,j) \vee M(i-1, j-T_i) & \text{if } i>0, i>0 \text{ and } j-T_i \geq 0 \\ M(i-1,j) & \text{if } j>0, i>0 \text{ and } j-T_i < 0 \end{cases} \Big\} \text{ recursive calls}$$

Bottom-up dynamic programming algorithm

```
Function DP (i,j):
    for i = 0 to n do
        M[i,0] = true
    for j = 1 to C
        M[0,j] = false
    else
        for i = 1 to n do
            for j = 1 to C do
                if j - T[i] ≥ 0 then
                    M[i,j] = M[i-1,j] or M[i-1,j-T[i]]
                else
                    M[i,j] = M[i-1,j]
    return M[n,C]
```

TIME COMPLEXITY → $O(nc)$

MINIMUM SPANNING TREE

- Dado um grafo conexo não dirigido (there is a path from any point to any other point)
  uma spanning tree é um subgrafo (árvore) que liga todos os vértices do grafo (undirected graph in which any two vertices are connected by exactly one path (acyclic graph))
- A soma de todos os pesos dos arcos é mínima
- O algoritmo de Kruskall cria uma MST, juntando gradualmente várias sub-árvores, cada uma sendo uma sub solução ótima da MST

KRUSKAL

1) Cada vértice é uma árvore com um elemento só

MakeSet(v)

2) Em cada iteração, o algoritmo junta duas sub-árvores através do arco livre com peso mais baixo

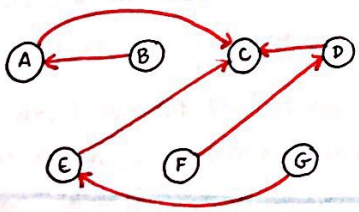
A = A + {(u,v)}
Union(u,v)

3) A função **FindSet(u)** devolve um identificador da árvore que contém o vértice u

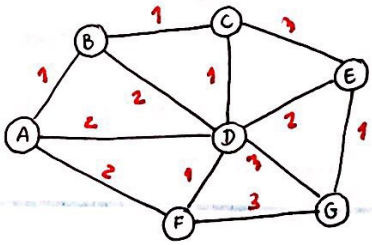
MSTKruskal(G, w)

A = {} ⟶ A é a MST a retornar
for each vertex v in G
  MakeSet(v) ⟶ cada nó é uma MST só com o próprio nó
sort edges of E into non-decreasing order by weight w
for each edge (u,v) in G, in non-decreasing order of weight ⟶ escolhe o edge livre com menor peso no grafo
  if FindSet(u) != FindSet(v) { ⟶ se não forem nós da mesma árvore
    A = A + {(u,v)} ⟶ junta os dois nós à solução MST
    Union(u,v) ⟶ une os dois conjuntos de MST que possuiam os nós u e v
return A

MAKE SET Each node is an element of singleton set & is the root



Merge sets that are connected by an edge ⇒ connect the roots



A disjoint set is a data structure which keeps track of all elements that are separated by a number of disjoint (not connected) subsets

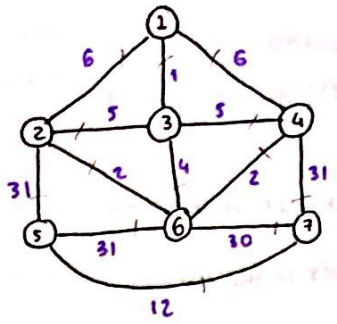
Two important functions

- Find ⇒ helps to determine which subset a particular element belongs to (or if the same element is in more than one subset)
- Union ⇒ helps to check whether a graph is cyclic or not
  connects / joins two subsets

PATH COMPRESSION

↳ instead of connecting only the root to the other root when joining two subsets, connect all nodes of the smallest subset directly to the root of the other subset

Consider the following graph

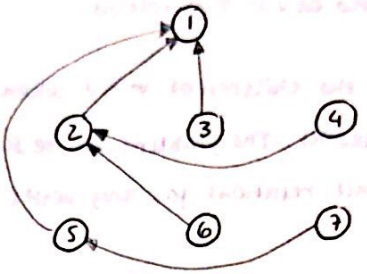


Draw its minimum spanning tree, as well as the graph of the union-find data structure, without path compression, using Kruskall algorithm

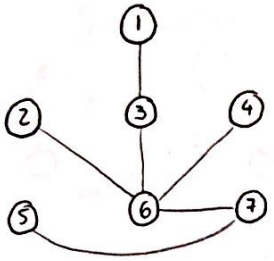
Always connect the root of the tree with the smallest height to the root of the tree with largest height, and, in case of a tie, choose as root the node with the smallest label

Sorted edges: ~~(1,3)~~ ~~(2,6)~~ ~~(4,6)~~ ~~(3,6)~~ ~~(2,5)~~ ~~(3,4)~~
(1,2) ~~(1,6)~~ ~~(5,7)~~ ~~(6,7)~~ (2,5) (4,7) (5,6)

Union-find data structure



Minimum spanning tree



Write the tail recursive version of the algorithm below, as well as its first call for an arbitrary non-negative number

```
Function real(n)
  if n=0 then
    return 0.0
  else
    return 1.0 + real(n-1)
```

Recursive methods are either
- Tail recursive: recursive call is the last statement
- Nontail recursive

Advantages of tail recursive methods
- Tail recursive can be transformed into iterative
  ⇒ gain in efficiency
- Some compilers can detect tail recursion to optimize code

```
Function real(n, res)
  if n=0 then
    return res
  else
    return real(n-1, r+1.0)
```

real(5, 0.0)
↳ real(4, 1.0)
↳ real(3, 2.0)
↳ real(2, 3.0)
↳ real(1, 4.0)
↳ real(0, 5.0)
↳ 5.0

A non tail recursive method for computing the factorial of n

```
Function fact(n)
  if n ≤ 1 then
    return 1
  else
    return n · fact(n-1)
```

A tail recursive method fact(n, 1) is the first call

```
Function fact(n, 1)
  if (n ≤ 1) then
    return res
  else
    return fact(n-1, n·res)
```

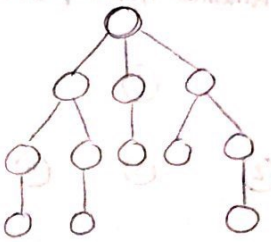
Consider the following problem

Given a tree $T(V,E)$, color as many vertices in T as possible, without coloring any pair of adjacent vertices (but 2 adjacent vertices are allowed to have no color)

In order to solve this problem, we consider 2 subproblems at each vertex $v \in V$

- $A(v)$ is the maximum number of coloured vertices in the subtree rooted at v, if v is not colored
- $B(v)$ is the maximum number of colored vertices in the subtree rooted at v if v is colored

$C(v) \rightarrow$ the children of $v \in V$ when traversing T from the root r to the leaves. The problem can be solved by the two following recurrence relations for any vertex $v \in V$



Quando v não está colorido, o nº max de vértices coloridos na sua subtree é dado, para cada filho de v, pelo somatório do valor máximo entre colorir ou não colorir o filho

$$A(v) = \sum_{u \in C(v)} \max\{A(u), B(u)\}$$

$A(u)$: don't color the child u — $B(u)$: color the child u

$$B(v) = (1) + \sum_{u \in C(v)} A(u)$$

(1): v is colored — $A(u)$: the sum of all the nodes further down the tree when the children of v aren't colored — then v can't be colored

The optimal value is given at the root r with $\max(A(r), B(r))$

Show that the calculation of $B(v)$ for any $v \in V$ is correct by using an optimal substructure argument

Let S be the optimal value for the problem above, for a node $v_i \in V$

$C(v)$ is the group of children of v

If the optimal solution for node v determines that v is colored, then, for all $u \in C(v)$, u can't be colored.

If we remove v from S, we are left with

Write the pseudo code of an algorithm that explores the two recursions above. Assume that the algorithm starts from the root r and traverses the tree in a depth-first search fashion. Give the first call of the algorithm for a tree T and a root r. In addition, discuss the time complexity of your approach

For each node v choose max value between

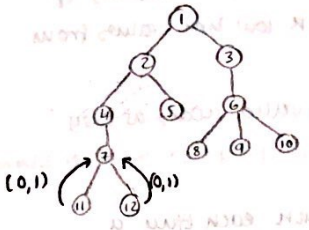
$$A(r) = \sum_{u \in C(v)} \max\{A(u), B(u)\}$$

$$B(v) = 1 + \sum_{u \in C(v)} A(u)$$

Function Color (T, v)

```
if C(v) = ∅
    A = 0 , B = 1
    return (A, B)
Else
    A = 0
    B = 1
    foreach u ∈ C(v)
        (a,b) = color (T, u)
        A = A + max(a, b)
        B = B + a
    return (A, B)
```

1ª chamada ⇒ Color(T, v,)



Strongly connected components

↳ Um grafo dirigido é uma componente fortemente conexa se existe um caminho entre cada nó & os restantes no grafo & a componente tem o maior tamanho possível

DFS a partir de um nó arbitrário

↳ um nó é raiz de uma componente fortemente conexa se o valor de dfs & low são iguais

Function Tarjan

```
stack S = {}
List scc = {}
for v in G
    if (dfs[v] == -1)
        tarjanAlgorithm (v, SCC, S)
return Scc
```

Function Tarjan Algorithm (v, scc, s)

```
dfs[v] = index
low[v] = index
index ++
s.push (v)
foreach (v,w) in G
    if dfs[w] = -1
        tarjan Algorithm (w, scc, s)
        low[v] = min (low[v], low[w]
    else if s.contains (w)
        low[v] = min (low[v], dfs[w])
if low[v] = dfs[v]
do
    w = s.pop()
    component.add (w)
while (w != v)
Scc.add(component)
```

Low-link value

↳ the smallest (lowest) node id reachable from that node when doing a DFS (including itself)

- all nodes which have the same low link value belong to the same strongly connected component

FLAW

Depending on where the DFS starts & which edges are visited, the low link values could be wrong.

The stack invariant

To cope with the random traversal order of the DFS, Tarjan's algorithm maintains a set (often a stack) of valid nodes from which to update low-link values from

Nodes are added to the stack of valid nodes as they are explored for the first time

Nodes are removed from the stack each time a complete SCC is found

Low-link value update condition

If u & v are nodes in a graph and we're currently exploring u, then our new low-link update condition is that:

↳ to update node u's low-link value to node v's low-link value, there has to be a path of edges from u to v & node v must be on the stack

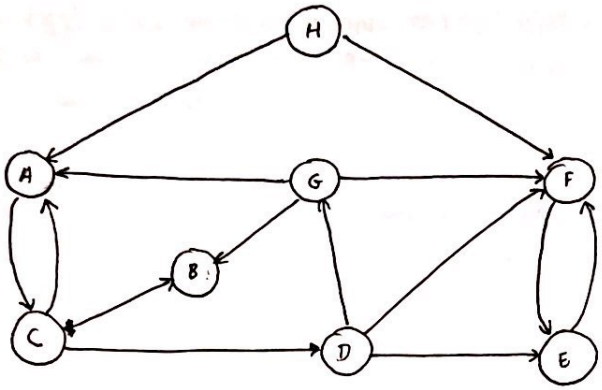
Tarjan's algorithm overview

don't stop until all the nodes are visited

1) Mark the ID of each node as unvisited
2) Start DFS. Upon visiting a node, assign it an ID & a low-link value. Also mark & current node as visited and add them to a seen stack
3) On DFS callback, if the previous node is on the stack, then min the current node's low-link value with the last node's low-link value → this allows low-link values to propagate throughout cycles.
4) After visiting all neighbours, if the current node started a connected component, then pop nodes off stack until current node is reached → this happens if the id equals its low link value

Find the strongly connected components of the following graph using Tarjan algorithm. Report the DFS tree(s) starting from vertex A & traversing the graph following the alphabetic order of the vertices labels. In addition, report the strongly connected components on the box below, ordered by the time they are found in the Tarjan algorithm.

Madalena Santos

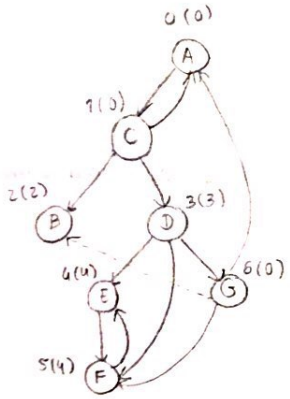


stack

H

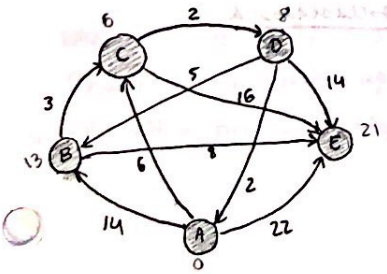
component G, D, C, A

scc {B}, {F, E}, {G, D, C, A}, {H}



RECURSO 2018

Find a shortest path in the left-hand graph between vertex A & vertex E using Dijkstra's algorithm. Draw the arcs that belong to the path in the right-hand graph. Fill in the table with the visited vertices, ordered according to the visiting order of Dijkstra algorithm, & with the shortest distance from vertex A to every vertex.

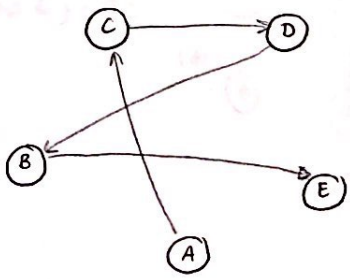


Vertices: A C D B E

Distance: 0 6 8 13 21

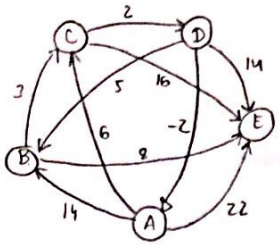
Function Dijkstra (G, source, target)

```
for each vertex v in G
    dist[v] = infinity
dist[source] = ∅
Q has the set of all nodes in G
while Q is not empty
    u = vertex in Q with smallest dist A, C, D, E
    remove u from Q
    if u = target
        break
    for each arc (v, u) in G
        if dist[v] > dist[u] + dist_between(v, u)
            dist[v] = dist[u] + dist_between(v, u)
return dist
```



6 + 2 + 5 + 8 = 21

Consider that the arc between vertex D & vertex A in the graph above has weight -2. Assuming that a path can contain repeated vertices, is the shortest path found in the previous exercise still valid after this transformation?

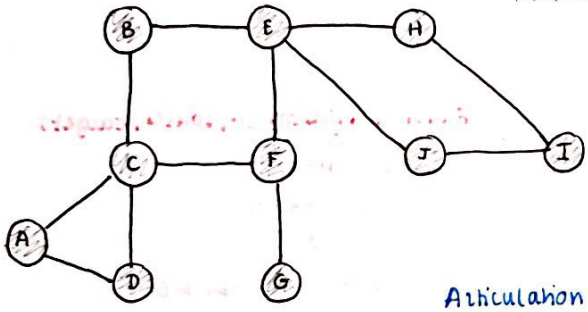


O peso do ciclo não é negativo, embora o custo da aresta o seja

O caminho continua a ser válido, pois nunca passa de D → A

Find the articulation points in the following graph. Justify your answer by reporting the DFS tree starting from node E, choosing the vertices for traversal in alphabetic order of the labels, & by explicitly writing the final values for dfs & low at each vertex. In addition, report the articulations points in the box below, ordered by the time they are found in the DFS tree traversal, as well as the criterion used to identify each point.

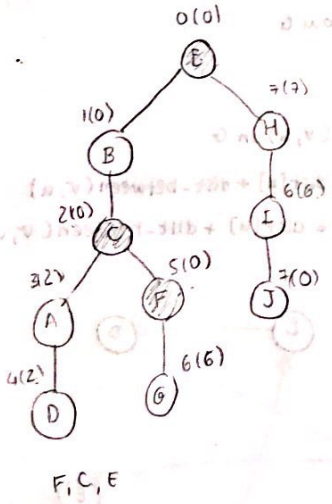
low[child] ≥ dfs[parent]



- Fazer um DFS
- Se o low[child] ≥ dfs[node], então node é um AP

Articulation points

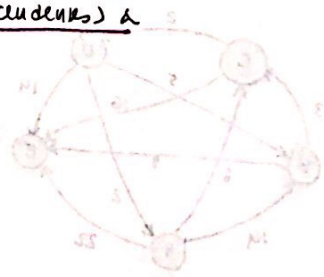
- G é um grafo conexo não dirigido
- um P.A. de G é um vértice cuja remoção torna G um grafo desconexo



F, C, E

Seja G_t a árvore DFS de um grafo

- A raiz r de G_t é um ponto de articulação se tem pelo menos dois filhos
- Um vértice v (não raiz) de G_t é um ponto de articulação se & só se v tem um filho w em G_t, tal que não existe nenhuma ligação (back edge) entre w (ou descendentes) a um antecessor de v.



dfs[v] é o índice da travessia dfs no vértice v

$$low[v] = \min \begin{cases} dfs[v] \\ dfs[x_i] \\ low[w_i] \end{cases}$$

- $dfs[v]$ → ancestrais de v com uma ligação a v
- $dfs[x_i]$
- $low[w_i]$ → filhos de v

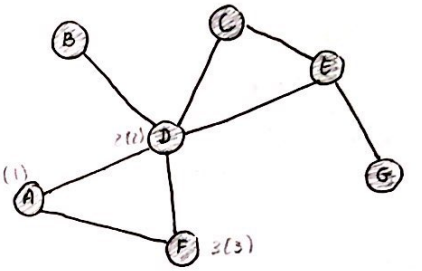
menor índice de travessia dfs de um nó que pode ser alcançado por v

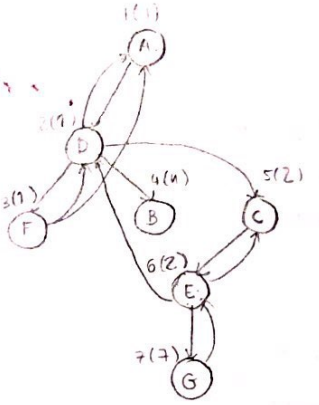
→ Um vértice v (não raiz) é um ponto de articulação se e só se tiver um filho w tal que

$$low[w] \geq dfs[v]$$

procedure Art(v)

```
low[v] = dfs[v] = time = time + 1
foreach (v, w) in E do
    if (dfs[w] = -1) then
        Art(w)
        low[v] = min{low[v], low[w]}
        if (dfs[v] = 1 & dfs[w] != 2) then
            "v is an articulation point"
        if (dfs[v] != 1) & (low[w] >= dfs[v]) then
            "v is an articulation point"
    else if (w is not parent of v)
        low[v] = min{low[v], dfs[w]}
```





E is an articulation point

D is an articulation point

Consider the following recurrence relation

Let $s_1, \ldots, s_n$ & $T_1, \ldots, T_m$ be two sequences of n & m chars, we define $A(i,j)$, $0 \leq i \leq n$ $0 \leq j \leq m$ as follows

$$A(i,j) = \begin{cases} j & \text{if } i = \emptyset \\ i & \text{if } j = \emptyset \\ \max\{A(i-1,j)+1, A(i,j-1)+1, A(i-1,j-1)+d\} & \text{if } i > \emptyset \ \& \ j > \emptyset \end{cases}$$

where $d = 1$ if $s_i = T_j$ & $d = 0$ otherwise

Pseudo-code of a bottom up DP algorithm to find the value for $A(n,m)$ & discuss its time complexity

if $i>0$ & $j>0$ $\Rightarrow A(i,j) = \max \begin{cases} A(i-1,j)+1 & \text{] value directly above} \\ A(i,j-1)+1 & \text{] value on the left} \\ A(i-1,j-1)+d & \text{] value in the left upper corner} \end{cases}$

$d = 1$ if $S_i = T_j$
$d = 0$ otherwise

Function DP()

```
for j=0 to m
  A[0,j] = j
for i=0 to n
  A[i,0] = i
for i=1 to n
  for j=1 to m
    if S[i] = T[j]
      d ← 1
    else
      d ← 0
    A[i,j] = max{A[i-1,j]+1, A[i,j-1]+1, A[i-1,j-1]+d}
return A[n,m]
```

Time complexity
O(nm) because of the loop

→ Posso acrescentar outro parâmetro?

Write a tail recursive function Find (A, n) in pseudo code that computes the minimum element of an array A that contains n positive integers. Assume that you cannot create neither global nor local variables in your program.

first call is: Find (A, n, +∞)

```
Function Find(A, n, min)
  if (A[n-1] < min)
    min = A[n-1]
  return Find(A, n-1, min)
```

BASE CASE

```
if (n = 0)   // we have finished the search
  return min
if (min = 0) // can't go lower than this
  return min
```

~~Function Find (A, n, min)~~
~~if (A[n-1] < min)~~
~~min = A[n-1]~~

```
Function Find(A, n, min)
  if n = 0
    return min
  if min = 0
    return min
  if (A[n-1] < min)
    min = A[n-1]
  return Find(A, n-1, min)
```

min = ~~2999~~ 1081
n = ~~8432~~ 5
A = 3, 5, 1, 8, 10

Consider the following problem

Given a grid of size $n \times m$, count the number of paths from the top left corner to the bottom right corner of the grid, assuming that only right or down directions are allowed in the grid

(A) Let $M(i,j)$, $1 \leq i \leq n$, $1 \leq j \leq m$ be the total number of paths that each position (i,j) in the grid. The value of $M(i,j)$ can be found by the following recurrence relation

$$M(i,j) = \begin{cases} 1 & \text{if } i=1 \text{ or } j=1 \\ M(i-1,j) + M(i,j-1) & \text{otherwise} \end{cases}$$

Show that the calculation of $M(i,j)$ for any i & j is correct

BASE CASE

Se $i=1$ ou $j=1$, só existe um caminho ← direita para baixo / baixo para direita

HIPÓTESE DE INDUÇÃO

Assumimos que $M(i,j) = M(i-1,j) + M(i,j-1)$ está correto

PASSO INDUTIVO

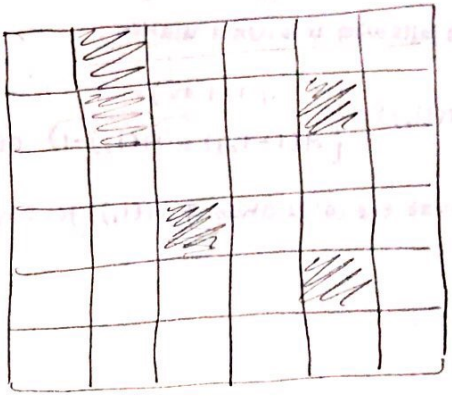
$$M(i+1, j+1) = M(i+1-1, j+1) + M(i+1, j+1-1)$$
$$= M(i, j+1) + M(i+1, j)$$

The following pseudo-code describes a bottom up dynamic programming algorithm that solves the problem above. Consider now that some positions cannot be visited, that is, they are forbidden. Let function forbidden(i,j) return true if position(i,j) is forbidden & false otherwise

Explain how you would modify the pseudo code bellow to count the total number of paths that do not go through the forbidden positions

```
Function Count(n,m)
  for i=1 to n do
    M[i,1]=1
  for j=1 to m do
    M[1,j]=1
  for j=2 to m do
    for i=2 to n do
      M[i,j]=M[i-1,j]+M[i,j-1]
  return M[n,m]
```

if forbidden(i,1) ⇒ M[i,1]=0
if M[i-1,1]=0 ⇒ M[i,1]=0
if forbidden(1,j) ⇒ M[1,j]=0
the same goes here
if forbidden(i,j) ⇒ M[i,j]=0



RECURSO 2017

Give the time complexity of the following algorithm that computes the minimum absolute difference between two integers in a list, & justify your answer using the Master Theorem. Assume that S = (S[1], ..., S[n]) is a non-ordered list of n distinct integers. In addition, assume that all arithmetic operations as well as abs take constant amount of time, and that the following functions take linear amount of time with respect to the number of elements in S: min(S), max(S), median(S), extract ≤ (S,K) & extract > (S,K) (returns the elements in S that are smaller than or equal to S[K], & larger than S[K], respectively)

```
Function ClosestPair(S)
  m = |S|
  if m=2 then
    d = ∞
  elseif m=2 then
    d = abs(S[2]-S[1])
  else
    K = median(S)
    S1 = extract≤(S,K)
    S2 = extract>(S,K)
    d1 = ClosestPair(S1)
    d2 = ClosestPair(S2)
    d = min(d1,d2,min(S2)-max(S1))
  return d
```

Master Theorem

→ a is the number of subproblems at each recursive step
→ b is the size of each subproblem
→ d is the cost of the base call
→ n^c is the cost of each recursive step

Let $a \geq 1$, $b > 1$, $d \geq 0$

$$T(n) = \begin{cases} aT(\frac{n}{b}) + n^c & \text{if } n > 1 \\ d & \text{if } n = 1 \end{cases} \Rightarrow T(n) = \begin{cases} \mathcal{O}(n^c) & \text{if } \log_b a < c \\ \mathcal{O}(n^c \log n) & \text{if } \log_b a = c \\ \mathcal{O}(n^{\log_b a}) & \text{if } \log_b a = c \end{cases}$$