GESTIONE DI INSIEMI DISGIUNTI (UNION-FIND)

Cap. 21.1 21.3

- Una struttura dati Z che rappresenta una suddivisione in insiemi disgiunti (i.e. una partizione) di $\{0, 1, 2, \dots n-1\}$

E.g. $\{1,3,7\}\ \{5,2\}\ \{4,0,8\}\ \{6\}$ è una partizione di $\{0,\dots,8\}$

- Le operazioni supportate dovrebbero essere almeno

componente di dim 1

1. Z = MAKESET(n) produce la partizione banale $\{0\}\ \{1\}\ \{2\}\ \dots\ \{n-1\}$

2. UNION(Z, x, y) unisce gli insiemi contenenti x e y se distinti.

E.g. $Z = \{1,3,7\}\ \{5,2\}\ \{4,0,8\}\ \{6\}$

↓ UNION(Z, 7, 0)

$Z = \{1,3,4,0,7,8\}\ \{5,2\}\ \{6\}$

↓ UNION(Z, 3, 1)

$Z = \{1,3,4,0,7,8\}\ \{5,2\}\ \{6\}$

3. Findset(Z, x) produce un valore tale che

$$\text{Findset}(Z,x) = \text{Findset}(Z,y)$$

se e solo se x e y sono nello stesso insieme

Tipicamente Findset(Z, x) restituisce il rappresentante dell'insieme che contiene x

per esempio

$Z = \{1,3,7\}\ \{5,2\}\ \{4,0,8\}\ \{6\}$

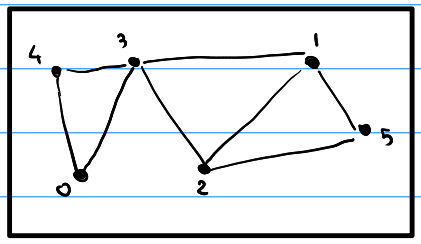
Findset(Z, x): 3 3 3 | 2 2 | 0 0 0 | 6

Applicazione per Kruskal

L'insieme dei vertici è identificato con $\{0,\dots,n-1\}$

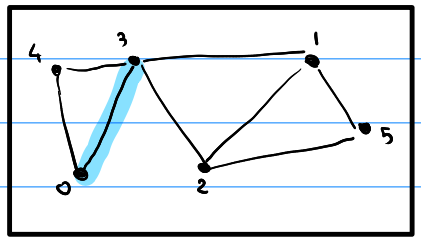
```
1  def Kruskal(n,E,W):
2      # Ordina rispetto ai pesi in
3      # modo non decrescente
4      E = sorted(E,key=lambda e: W[e])
5      T = []
6      Z = MakeSet(n)
7      for u,v in E:
8          if FindSet(Z,u) != FindSet(Z,v):
9              T.append((u,v))
10             Union(Z,u,v)
11     return T
12
```

Componenti connesse quando non ci sono archi $\{0\},\{1\},\dots,\{n-1\}$

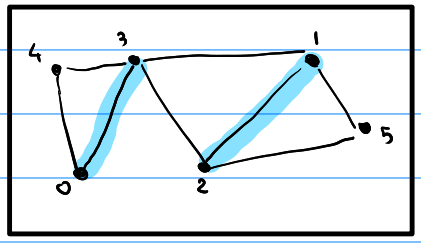


Ignoriamo i pesi degli archi e assumiamo che siano scelti in ordine crescente

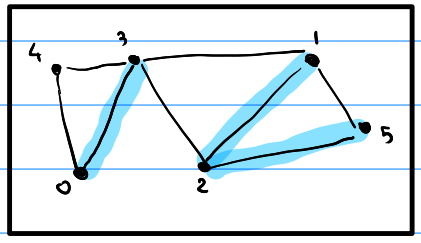
$\{0\}\ \{1\}\ \{2\}\ \{3\}\ \{4\}\ \{5\}$



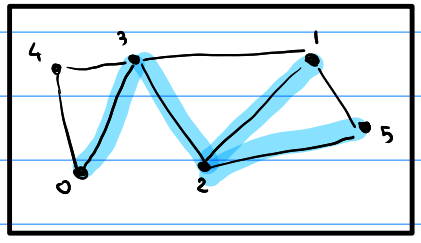
$\{1\}\ \{2\}\ \{0,3\}\ \{4\}\ \{5\}$



$\{1,2\}\ \{0,3\}\ \{4\}\ \{5\}$

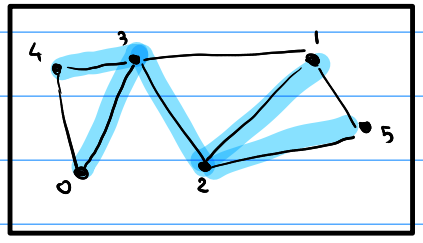


$\{1,2,5\}\ \{0,3\}\ \{4\}$



$\{1,2,0,5,3\}\ \{4\}$

$\{0,1,2,3,4,5\}$



PRIMA IDEA:

Z potrebbe essere un array dei rappresentanti

$Z[i] \rightarrow$ rappresentante dell'insieme che contiene i

MAKESET$(n) \rightarrow [0, 1, 2, 3, \ldots, n-1]$

FINDSET$(Z, i) \rightarrow Z[i]$

- Tuttavia l'operazione di UNION è costosa

$Z = \{0,1,2,3,4\}\ \{5,6,7,8,9\} \xrightarrow{\text{UNION}(Z,1,8)} \{0,1,2,3,4,5,6,7,8,9\}$

$[2,2,2,2,2,6,6,6,6,6] \longrightarrow [6,6,6,6,6,6,6,6,6,6]$

- ESERCIZIO Trovare una serie di operazioni UNION tale che ci vogliono $\Theta(n^2)$ operazioni per andare dalla partizione iniziale a quella finale (un singolo insieme)

Nota: potete assumere che tra i due rappresentanti venga scelto sempre quello di indice più alto

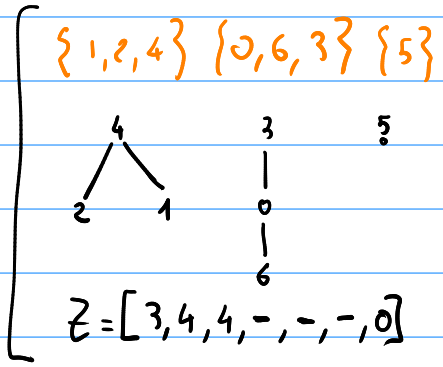
SECONDA IDEA

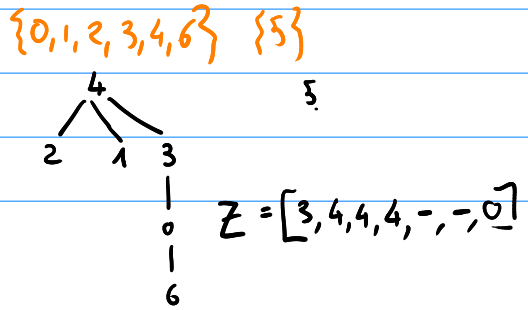
Z rappresenta un insieme di alberi. La radice è il rappresentante di ogni insieme

MAKESET$(n) \rightarrow [\text{NIL}, \text{NIL}, \text{NIL}, \ldots]$

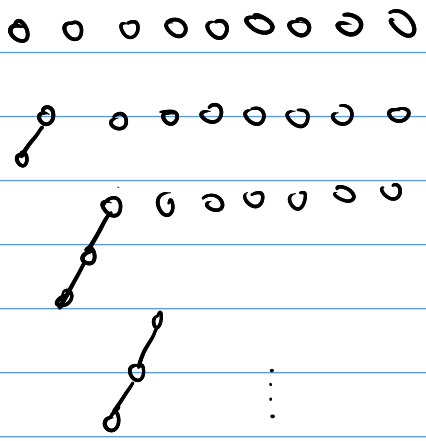
```
FindSet(Z, i):
    if Z[i] = NIL:
        return i
    else return FINDSET(Z, Z[i])
```

```
UNION(Z, x, y):
    a = FindSet(Z, x)
    b = FindSet(Z, y)
    if a ≠ b:
        Z[a] = b
```





In questa rappresentazione ogni chiamata a FindSet richiede di risalire l'albero per scoprire la radice. In certi casi questo può costare molto



- È possibile che una sequenza sfortunata di UNION possa creare alberi molto alti e sbilanciati
- Esempio
  UNION(Z,0,1)
  UNION(Z,0,2)
  UNION(Z,0,3)
  ⋮
- Ogni UNION(Z,0,i) costa i passi
- Ogni FINDSET(Z,0) costa t passi se eseguita dopo UNION(Z,0,t)

La struttura migliorerà molto usando due idee:

- UNIONE PER RANGO
- COMPRESSIONE DEI CAMMINI

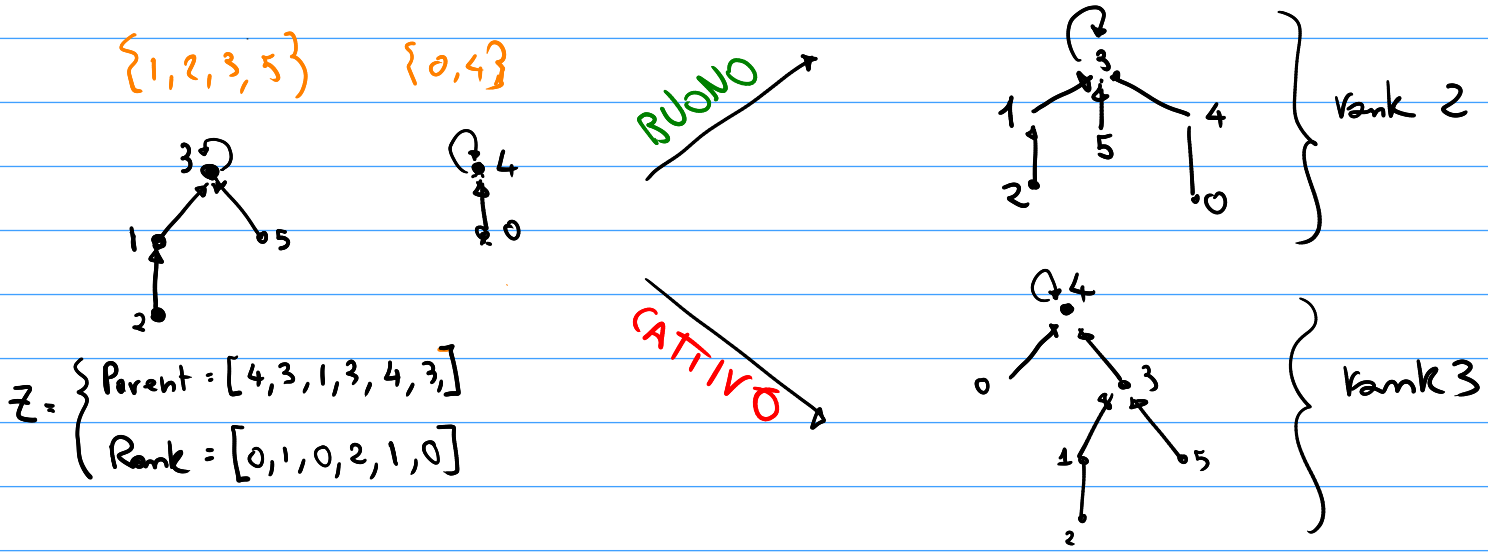
Le due idee sono indipendenti quindi le vediamo una alla volta

- **UNIONE PER RANGO**

Visto che il costo delle operazioni FINDSET è legato all'altezza dell'albero cerchiamo di utilizzare alberi meno "asimmetrici" e quindi, a parità di elementi memorizzati, più bassi.

$Z = \begin{cases} \text{Parent} & \text{array dei genitori che rappresenta gli alberi} \\ \text{Rank} & \text{array che da una MISURA dell'altezza dell'albero} \end{cases}$

OSS Per comodità le radici non avranno genitore NIL ma genitore uguale a se stesso

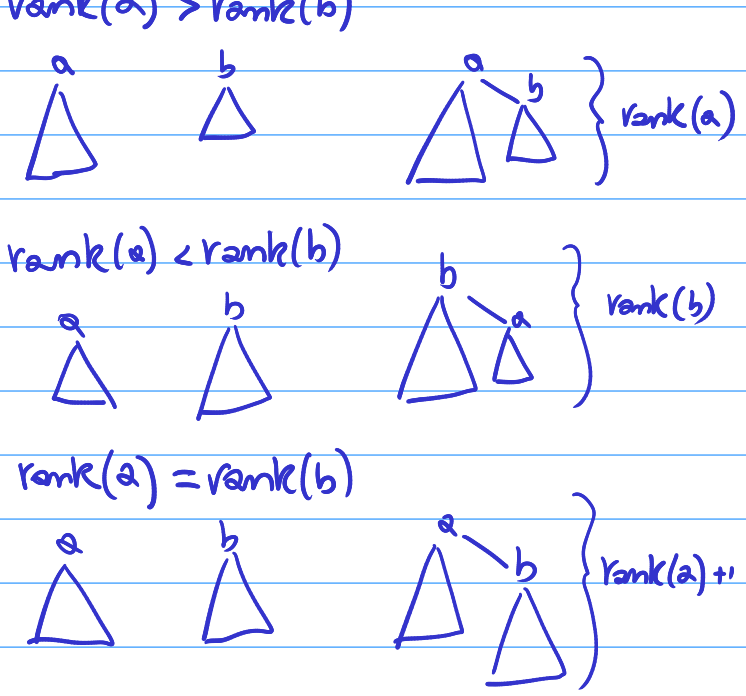


USIAMO RANK PER CERCARE DI NON FAR INCREMENTARE TROPPO L'ALTEZZA DEGLI ALBERI: METTO L'ALBERO PIÚ BASSO COME FIGLIO DI QUELLO PIÚ ALTO

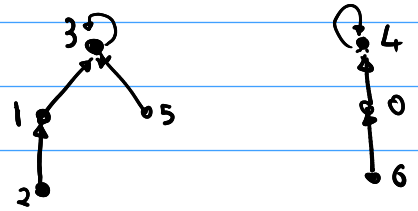
MAKESET(n): { Parent: [0,1,2,3,...,n-1]
Rank: [0,0,0,0,...,0]

```
findSet(Z,i):
    while Z.Parent[i] ≠ i
        i ← Z.Parent[i]
    return i
```

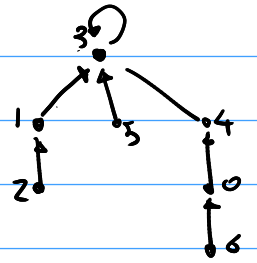
```
UNION(Z,x,y) =
    a = findSet(Z,x)
    b = findSet(Z,y)
    if a = b
        return
    if Z.Rank[a] > Z.Rank[b]
        Z.Parent[b] = a
    elif Z.Rank[a] < Z.Rank[b]
        Z.Parent[a] = b
    else:
        Z.Parent[b] = a
        Z.Rank[a] += 1
```



$\{1,2,3,5\}$ $\{0,4,6\}$



$z = \begin{cases} \text{Parent} = [4,3,1,3,4,3,0] \\ \text{Rank} = [1,1,0,2,2,0,0] \end{cases}$

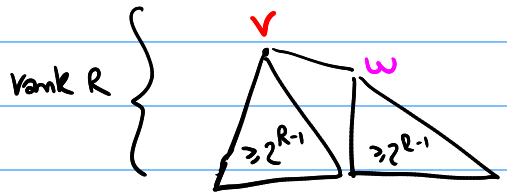


$z = \begin{cases} \text{Parent} = [4,3,1,3,3,3,0] \\ \text{Rank} = [1,1,0,3,2,0,0] \end{cases}$

Lemma Se v è
- radice di uno degli alberi
- ha rank R

allora l'albero di cui è radice contiene ALMENO 2^R elementi (compreso v)

dim • Per induzione su R
- Se $R=0$, $2^R=1$ e l'albero contiene v
- Se $R>0$, consideriamo il momento in cui v è diventato di rank R la prima volta. Caso else di UNION: v unito all'albero di un vertice w, e entrambi hanno rank $R-1$



Per ipotesi induttiva i sottoalberi radicati in v e w hanno entrambi almeno 2^{R-1} ciascuno. Quindi dopo l'unione, l'albero radicato in v contiene $\geq 2^R$ elementi.

Corollario (Esercizio 21.4-2)
Ogni nodo ha rango al più $\log_2 n$

Corollario (Esercizio 21.4-4)
m operazioni UNION/FINDSET hanno complessità $O(m \log n)$

Corollario
L'algoritmo di Kruskal con questa implementazione di UNION/FINDSET ha complessità $O(|E| \log |V|)$

ESERCIZIO 21.3-3

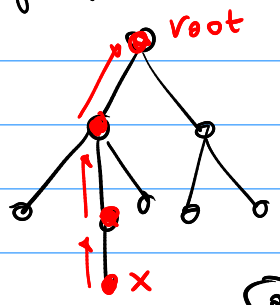
Assumete di aver fatto MAKESET(n),
adesso trovate una sequenza di UNION/FINDSET tali che la complessità (usando l'implementazione vista), impieghi $\Omega(m \log n)$ operazioni

(l'analisi dell'esercizio 21.4-4 non può essere migliorata

Vedere esempio Precalcolato

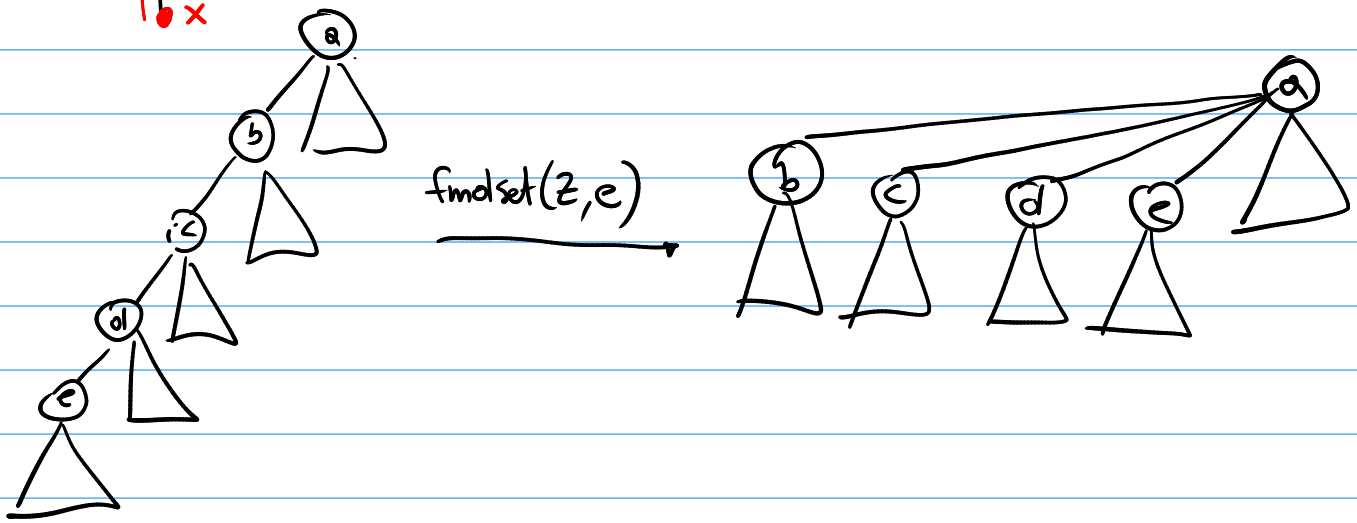
COMPRESSIONE DEI CAMMINI

Quando cerchiamo la radice di un albero contenente un nodo x, risaliamo lungo l'albero.



Non ha senso ripetere la ricerca:

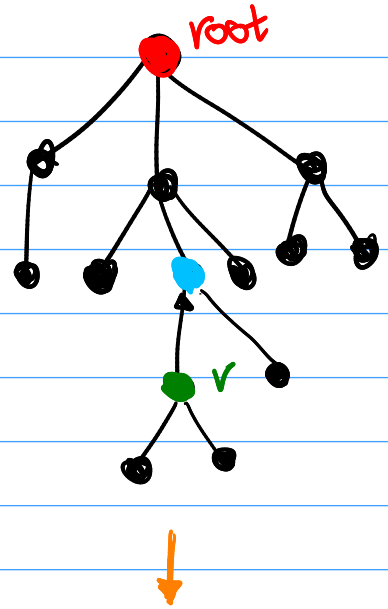
GLI INSIEMI NON VENGONO MAI DIVISI



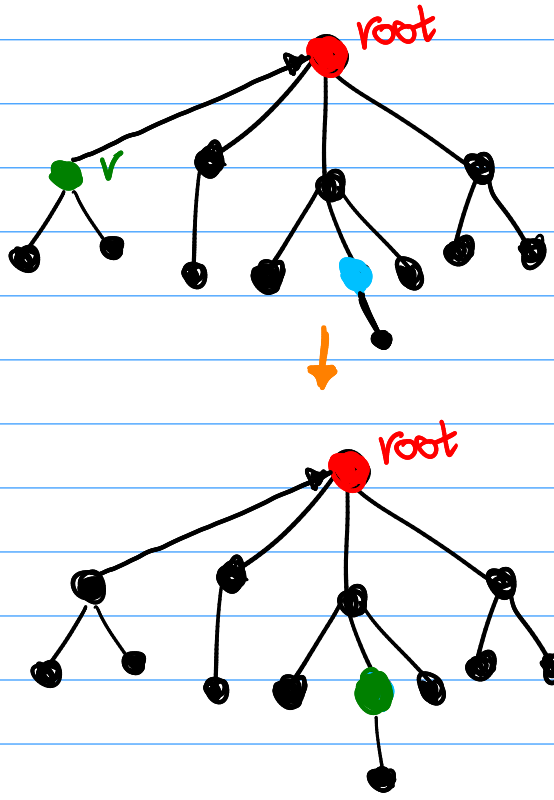
OSS Il rank non è più la lunghezza del cammino più lungo, ma è una STIMA IN ECCESSO DELLO STESSO

OSS Il lemma a Pag 6 rimane valido anche usando la compressione dei cammini (ricordate che il lemma è solo per i nodi radici di alberi)

```
findset(Z, v):
    p ← Z.Parent
    root ← v
    while root != P[root]:
        root ← P[root]
    while v != root
        parent ← P[v]
        P[v] ← root
        v ← parent
```



- Questa versione fa due cicli lungo il cammino da v a root
  Mentre la versione originale ne faceva uno.
- findSet è quindi sempre $O(h)$ dove h è la lunghezza del cammino
- Tuttavia la distanza tra v e root diminuisce nel tempo.



- Adesso vediamo l'esecuzione dell'algoritmo di Kruskal con questa implementazione di UNION / FIND

Vedere esempio precalcolato

EFFICIENZA DI UNION/FIND CON

- UNIONE PER RANGO
- COMPRESSIONE DEI CAMMINI

- La compressione dei cammini non peggiora i tempi di esecuzione, quindi MAKESET(n) seguito da m operazioni UNION/FINDSET costa $O(n) + O(m \log n)$

ESERCIZI Dimostrate le seguenti affermazioni riguardanti il Rank.

1. Rank misura l'altezza che avrebbe un elemento SE NON CI FOSSE LA COMPRESSIONE DI CAMMINI
2. Ogni elemento non radice ha rank strettamente minore di quello del genitore
3. Il rank di un elemento cambia solo finché è una radice. Quando non è più radice, rimane fisso
4. Usando il lemma a pag 6 si può dimostrare che per $R \geq 0$ ci sono $\leq \frac{n}{2^R}$ elementi di rank maggiore di R

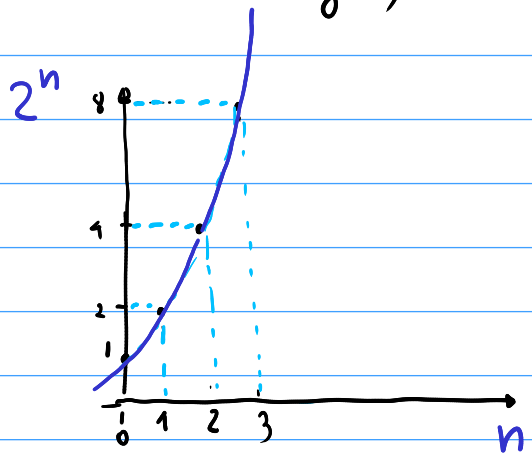
L'upper bound è molto debole rispetto alle vere performance di UNION/FINDSET con UNIONE PER RANGO e PATH COMPRESSION

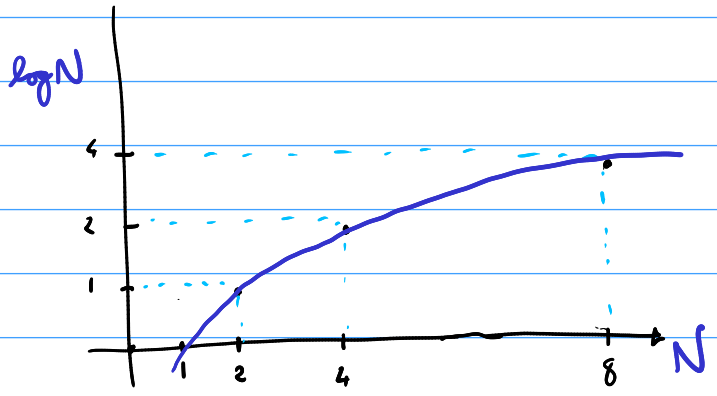
Thm Effettuare m operazioni di UNION/FINDSET dopo un MAKESET(n) costa

$$\Theta(m \cdot \alpha(n)) \quad \text{operazioni}$$

Dove $\alpha(n)$ è l'inverso della funzione di Ackermann

- La funzione 2^n cresce molto in fretta e quindi la sua INVERSA, ovvero $\log N$, cresce molto lentamente





- La funzione di Ackermann è una funzione che cresce in maniera ESTREMAMENTE RAPIDA (e la sua inversa è ESTREMAMENTE LENTA)

~ ci sono diversi modi di definire questa funzione, tutti più o meno con le stesse importanti proprietà. Vediamo la seguente definizione

- $A_0(x) = x + 1$
- $A_k(x) = A_{k-1}^{(x+1)}(x)$ per $k > 0$

Iterazione di funzione
$f^{(0)}(x) = x$
$f^{(i+1)}(x) = f^{(i)}(f(x))$
E.g. $f^{(4)}(x) = \underbrace{f(f(f(f(x))))}_{4}$

Prop
- $A_1(x) = 2x - 1$ $\forall x \geq 1$ intero
- $A_2(x) = 2^{(x+1)} \cdot (x+1) - 1$
- $A_3(1) = A_2^{(2)}(1) = A_2(A_2(1)) = A_2(7) = 2^8 \cdot 8 - 1 = 2047$
- $A_4(1) = A_3^{(2)}(1) = A_3(A_3(1)) = A_3(2047) = A_2^{(2048)}(2047)$

>> enormemente più grande >> $A_2(2047) = 2^{2048} \cdot 2048 - 1$

$>> 10^{616}$

[Il numero di atomi nell'universo è stimato grossolanamente $< 10^{82}$]

- Quindi, consideriamo la funzione

$$k \longmapsto A_k(1)$$

questa funzione cresce enormemente. La sua inversa è

$$\alpha(n) = \min_{k \geq 0} \{ A_k(1) \geq n \}$$

dunque

$$\alpha(n) = \begin{cases} 0 & \text{per } 0 \leq n \leq 2 \\ 1 & \text{per } n = 3 \\ 2 & \text{per } 4 \leq n \leq 7 \\ 3 & \text{per } 8 \leq n \leq 2047 \\ 4 & \text{per } 2048 \leq n \leq A_4(1) \end{cases}$$

- Quindi per qualunque input realistico, $\alpha(n) \leq 4$ e quindi union/findset si comporta quasi come se avesse complessità lineare.

ESERCIZIO
- Scrivete un programma che calcoli $A_k(x)$.
- Attenzione alla grandezza dei numeri!
- Fino a che valore di k riuscite a calcolare qualcosa?