Algorithm Theory

09 – Union-Find Data Structures

Dr. Alexander Souza

Winter term 11/12

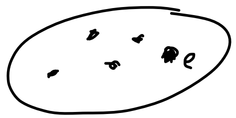
Union-find data structures

Problem:

Maintain a collection of disjoint sets while supporting the following operations:

e.make-set(): Creates a new set whose only member is *e*.

e.find-set(): Returns the set M_i containing *e*.

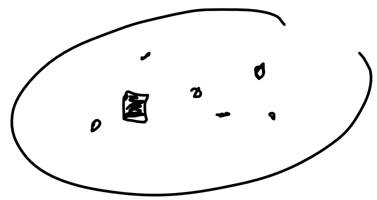


union(M_i, M_j): Unites the sets M_i and M_j into a new set.

Union-find data structures

Representation of set M_i :

M_i is identified by a **representative**, which is some member of M_i.



Union-find data structures

Operations using representatives:

e.make-set():

Creates a new set whose only member is e. The representative is e.

e.find-set():

Returns the name of the representative of the set containing e.

e.union(f):

Unites the sets M_e and M_f that contain e and f into a new set M and returns a member of $M_e \cup M_f$ as the new representative of M.

The sets M_e and M_f are then „destroyed".

Observations

- If n is the number of *make-set* operations and m the total number of *make-set*, *find-set* and *union* operations, then
  - $m >= n$
  - after $(n - 1)$ *union* operations, only one set remains in the collection

Application: Connected components

Input: graph $G = (V,E)$

Output: collection of the connected components of G

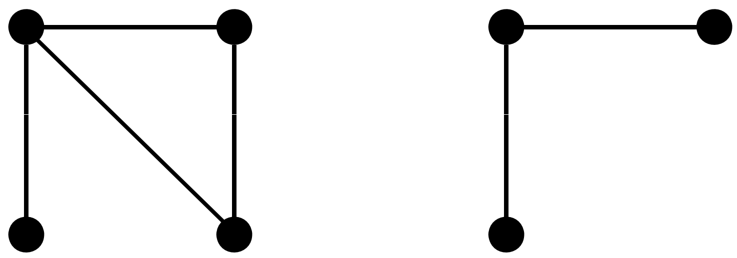
Algorithm: Connected-Components

for all *v* **in** *V* **do** *v.make-set()*

for all *(u,v)* **in** *E* **do**

  if *u.find-set()* $\neq$ *v.find-set()*

    then *u.union(v)*



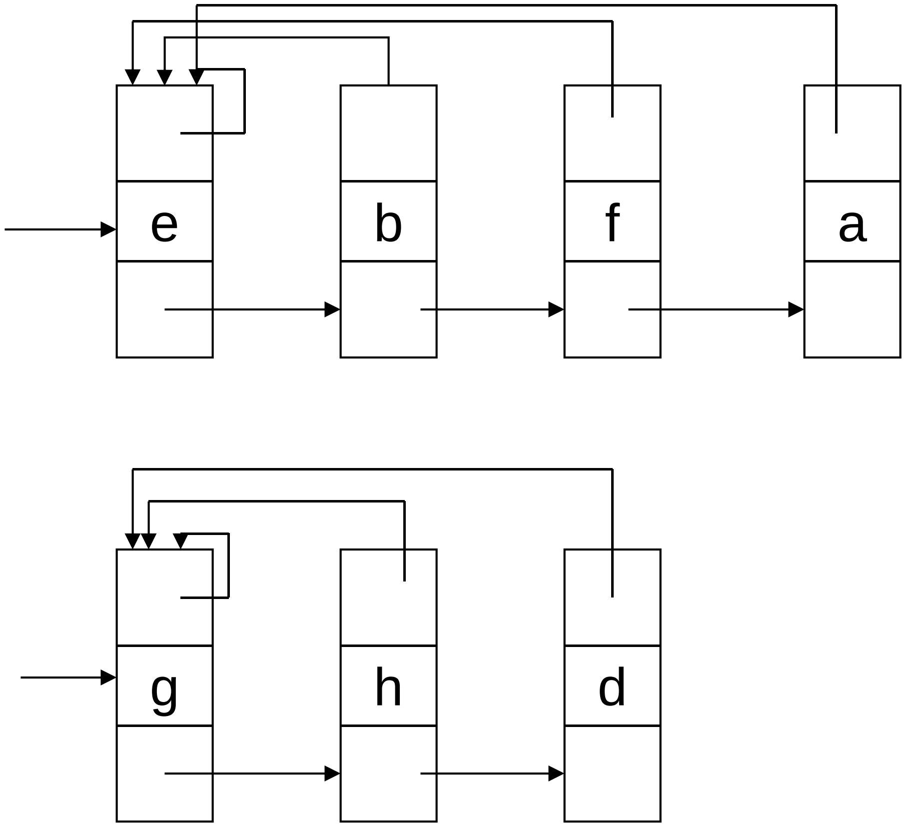
Same-Component *(u,v):*

  if *u.find-set() = v.find-set()*

  then return *true*

  else return *false*

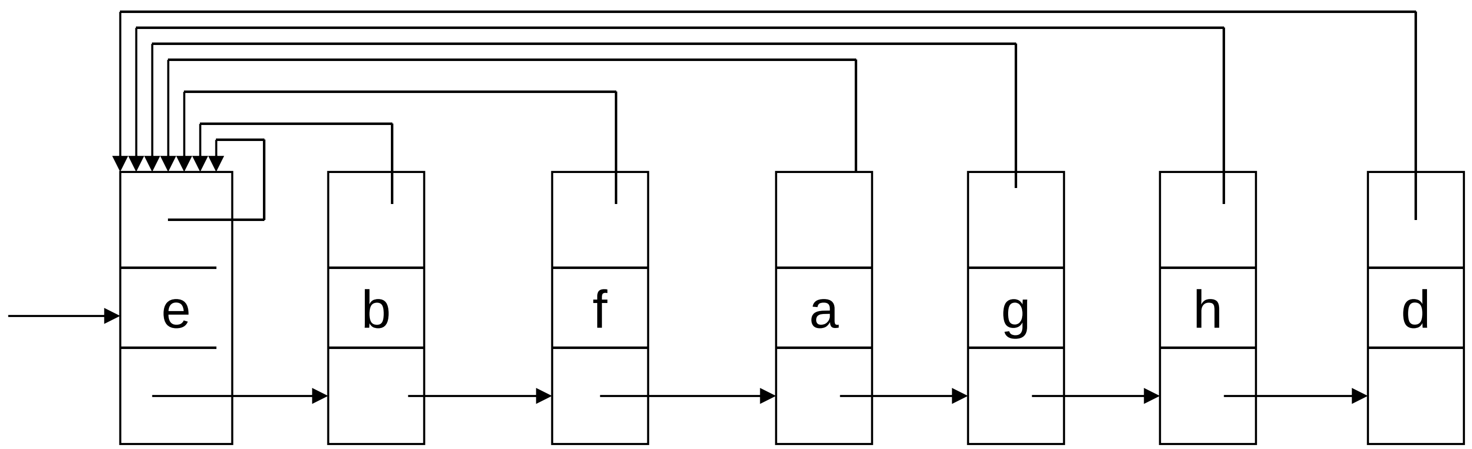
Linked-list representation



- *x.make-set()*
- *x.find-set()*
- *x.union(y)*

Linked-list representation

b.union(d)



„Bad“ sequence of operations

e_1*.make-set()*

e_2*.make-set()*

⋮

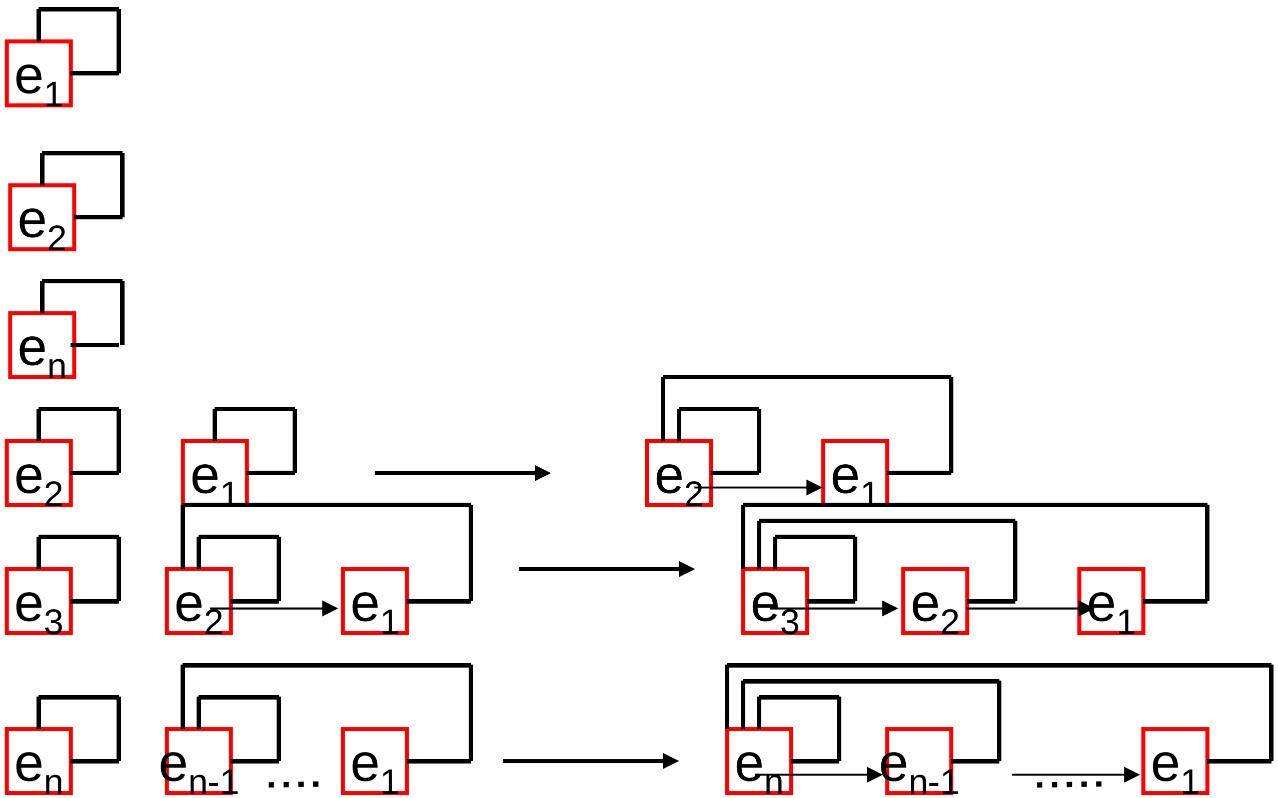
e_n*.make-set()*

e_2*.union(*e_1*)*

e_3*.union(*e_2*)*

⋮

e_n*.union(*e_{n-1}*)*



The longer list is always appended to the shorter list!

Pointer updates for the *i*-th operation $e_i.union(e_{i-1})$:

Running time of $2n - 1$ operations:

Improvement

Weighted-union heuristic

Always append the smaller list to the longer list.
(Maintain the length of a list as a parameter).

Theorem

Using the weighted-union heuristic, the running time of a sequence of *m* *make-set, find-set,* and *union* operations, *n* of which are *make-set()* operations, is *O(m + n log n)*.

Improvement: $O(m \cdot \alpha(m, n))$

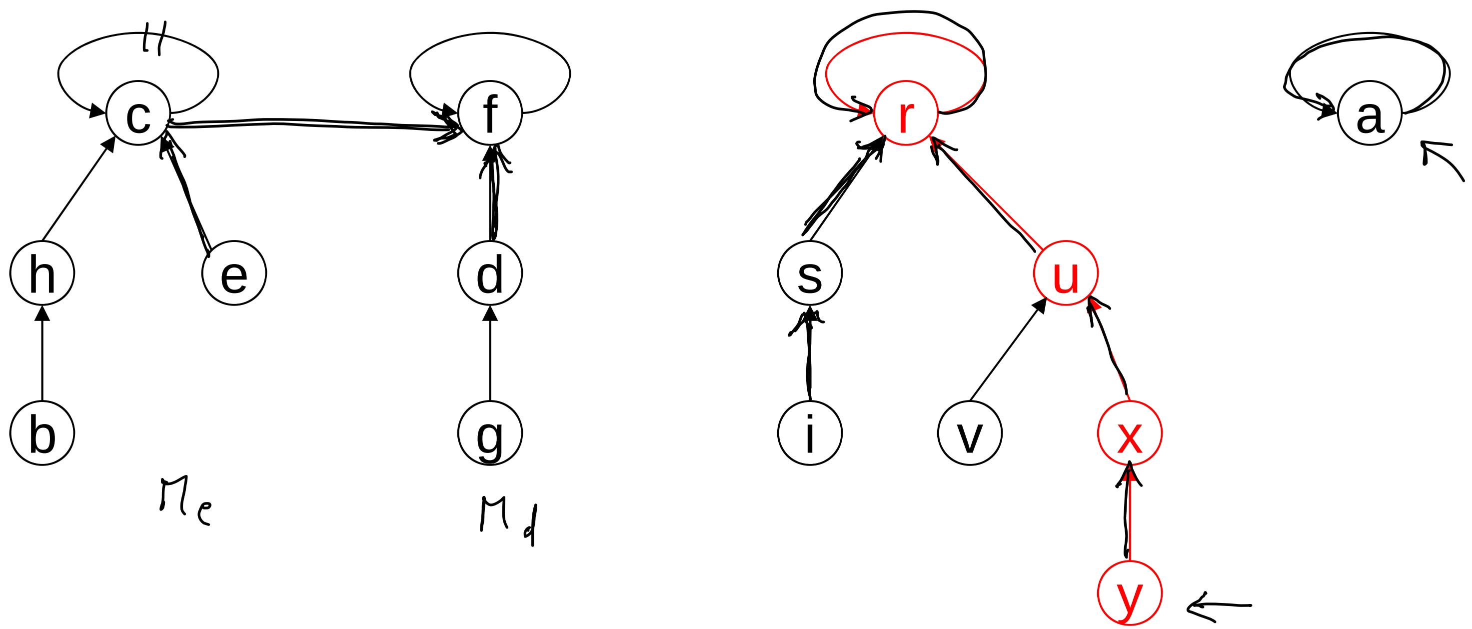
$\alpha(m, n)$ inverse Ackermann function

Proof

Consider element *e.*

Number of times *e*'s pointer to the representative is updated: log *n*

Disjoint-set forests

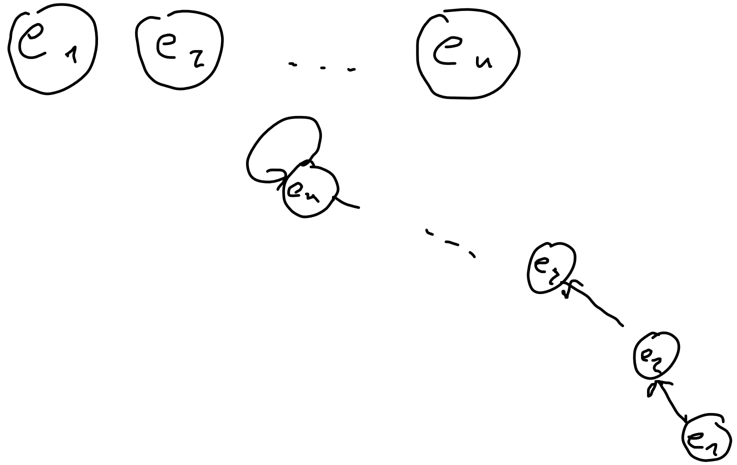


- *a.make-set()*
- *y.find-set()*
- *d.union(e):* Make the representative of one set (e.g. *f*) the parent of the representative of the other set.

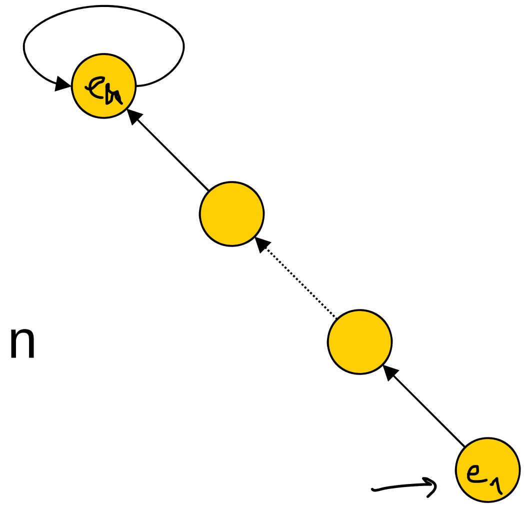
Example Bad Sequence

m = total number of operations ($\geq 2n$)

for $i = 1$ **to** n **do** e_i*.make-set( )*
for $i = 2$ **to** n **do** e_i*.union(*e_{i-1}*)*
for $i = 1$ **to** f **do** e_1*.find-set( )*



n-th step



Chain

running time of f *find-set* operations: $\Theta(f * n)$

linear ~ bad.

Union by size

additional variable:

e.size = (# nodes in the subtree rooted at *e*)

e.make-set()

1 *e.parent* = *e*

2 *e.size* = 1

e.union(f)

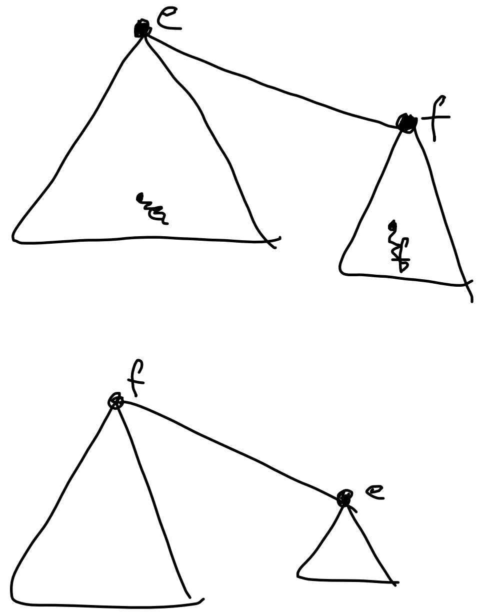
1 *link(e.find-set( ), f.find-set( ))*

Union by size

representatives

link(e,f)

```
if e.size ≥ f.size
    then f.parent = e
         e.size = e.size + f.size
    else /* e.size < f.size */
         e.parent = f
         f.size = e.size + f.size
```



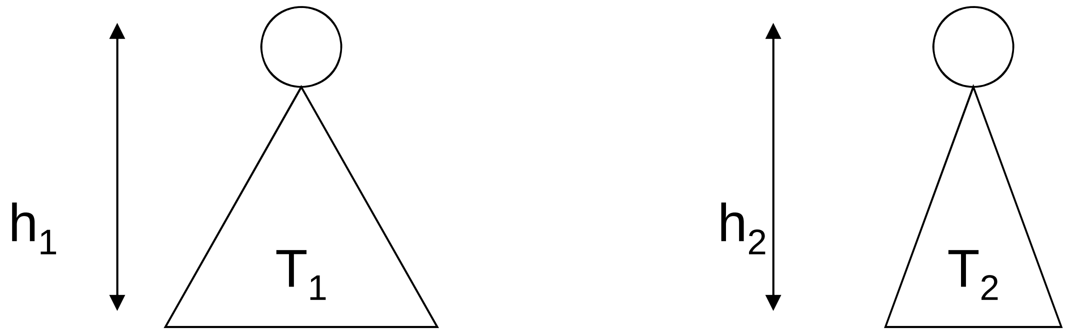
Union by size

Theorem

The method union-by-size maintains the following invariant:

A tree of height *h* contains at least 2^h nodes.

Proof



Base Case:

$h = 0$

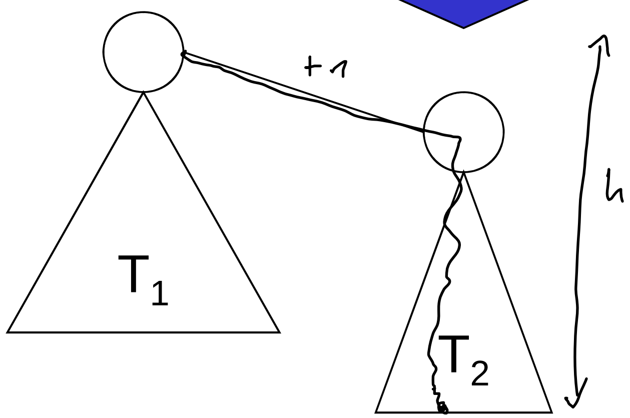
○ T

$g = 1 = 2^0 = 2^h$

$|T_1| \geq |T_2|$

$g_1 = |T_1| \geq 2^{h_1}$ $g_2 = |T_2| \geq 2^{h_2}$

T



$g = g_1 + g_2 \overset{?}{\geq} 2^h$

Union by size

h h_1

Case 1: The height of the new tree is equal to the height of T_1. $h = h_1$

$$g = g_1 + g_2 \geq g_1 \geq 2^{h_1} = 2^h \checkmark$$

Case 2: The new tree T has a greater height.

height of T: $h_2 + 1 = h$

$$\geq 2 \cdot g_2 \geq 2 \cdot 2^{h_2}$$

$$g = g_1 + g_2 \geq \cancel{2^{h_2} + 2^{h_2}} = 2^{h_2+1} = 2^h \checkmark$$

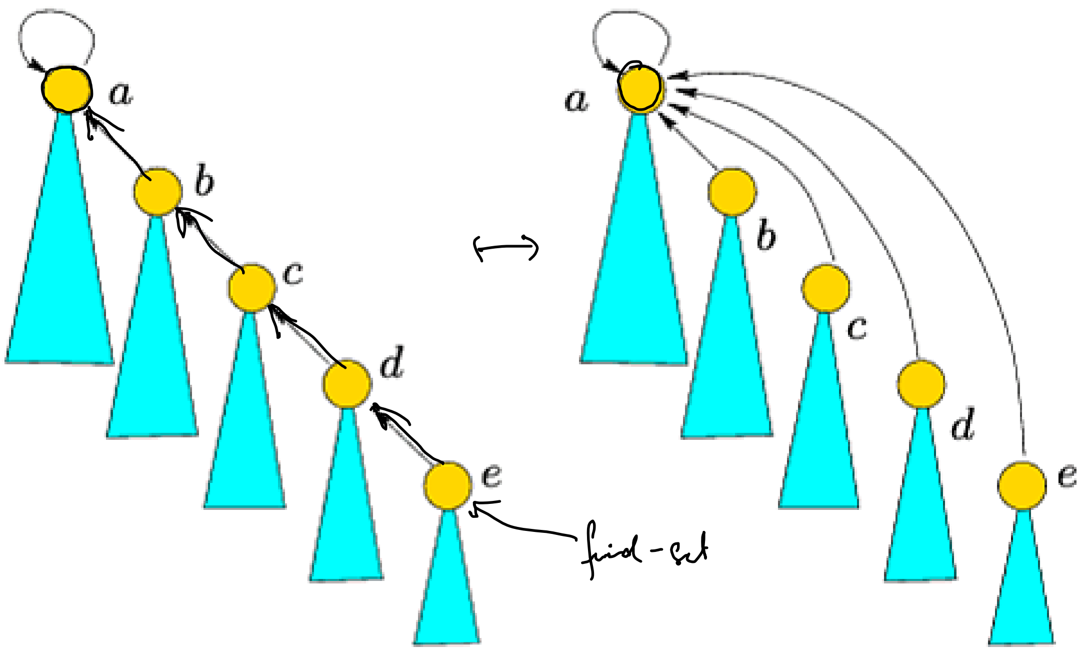
Consequence

Height is logarithmic

The running time of a *find-set* operation is O(*log n*), where *n* is the number of *make-set* operations.

Further improvement

Path compression during 'find-set' operations



Elegant recursive implementation

e.find-set()

```
if e ≠ e.parent                          // e is not root
     then e.parent = e.parent.find-set( )      // e.parent = representative
return e.parent
```

Analysis of the running time

m total number of operations,

f of which are *find-set* operations and
n of which are *make-set* operations
→ at most *n* – *1 union* operations

Union by size: without path compression

O($n + f \log n$)

find-set **operation with path compression:**

If $f < n$, $\Theta(n + f \log n)$

If $f \geq n$, $\Theta(f \cdot \log_{1+f/n} n)$

≥ 2

Analysis of the running time

Theorem (Union by size with path compression)

Using the combined *union-by-size* and *path-compression* heuristic, the running time of m disjoint-set operations on n elements is

$$\boxed{\Theta(m * \alpha(m,n)),}$$

where $\alpha(m,n)$ is the inverse of Ackermann's function.

Ackermann's function and its inverse

Ackermann's function $A(\cancel{m}, i, j)$

$A(1,j) = 2^j$ for $j \geq 1$

$A(i,1) = A(i-1,2)$ for $i \geq 2$

$A(i,j) = A(i-1, A(i, j-1))$ for $i,j \geq 2$

grows extremely fast

inverse of Ackermann's function

$$\alpha(m,n) = \min\{ i \geq 1 \mid A(i, \lfloor m/n \rfloor) > \log n \}$$

grows extremely slow

$n \sim 10^{80}$ $\alpha(m,n) \leq 4$

Ackermann's function and its inverse

$$A(i,\lfloor m/n \rfloor) \geq A(i,1)$$

$$A(2,1) = A(1,2) = 2^2 = 4$$

$$A(3,1) = A(2,2) = A(1, A(2,1)) = 2^4 = 16$$

$$A(4,1) = A(3,2) = A(2, A(3,1)) = A(2,16)$$

$$\geq 2^{2^{2^{2^2}}} = 2^{65536}$$

$$\alpha(m,n) \leq 4, \text{ for } n \text{ satisfying} \log n < 2^{65536}$$