

# **Algoritmi e Strutture Dati**

Schifano S. Fabio `schifano@fe.infn.it`

# Insiemi

Un insieme è una collezione di elementi distinti dello stesso tipo.

A differenza delle altre strutture dati, tipo le liste, gli elementi non sono dell'insieme non sono caratterizzati da una posizione nè sono ammesse copie di uno stesso elemento.

Un insieme può essere descritto in due modi diversi:

- ▷ elencando gli elementi dell'insieme, ad esempio:  $P = \{2, 4, 6, 8, \dots\}$
- ▷ tramite una relazione che ne caratterizza gli elementi, ad esempio:  $P = \{n \mid n = 2 * i, i \in \mathbb{N}\}$

Il numero degli elementi che compongono l'insieme é detto **cardinalità**.

Le operazioni fondamentali per gli insiemi sono:

- ▷ appartenenza
- ▷ intersezione
- ▷ unione
- ▷ inclusione
- ▷ ...

## Specifica degli Insiemi

- ▷ CREAinsieme:  $() \longrightarrow \text{insieme}$   
 $\text{CREAinsieme}() = A$   
Post:  $A = \phi$
- ▷ INSIEMEvuoto:  $(\text{insieme}) \longrightarrow \text{boolean}$   
 $\text{INSIEMEvuoto}(A) = b$   
Post:  $b = \text{True}$  se  $A = \phi$ ,  $b = \text{False}$  altrimenti
- ▷ APPARTIENE:  $(\text{tipoelem}, \text{insieme}) \longrightarrow \text{boolean}$   
 $\text{APPARTIENE}(x, A) = b$   
Post:  $b = \text{True}$  se  $x \in A$ ,  $b = \text{False}$  se  $x \notin A$
- ▷ INSERISCI:  $(\text{tipoelem}, \text{insieme}) \longrightarrow \text{insieme}$   
 $\text{INSERISCI}(x, A) = A'$   
Post:  $A' = A \cup \{x\}$ , se  $x \in A$  allora  $A' = A$
- ▷ CANCELLA:  $(\text{tipoelem}, \text{insieme}) \longrightarrow \text{insieme}$   
 $\text{CANCELLA}(x, A) = A'$   
Post:  $A' = A - \{x\}$ , se  $x \notin A$  allora  $A' = A$

▷ UNIONE: (insieme, insieme)  $\longrightarrow$  insieme

$$\text{UNIONE}(A, B) = C$$

$$\text{Post: } C = A \cup B$$

▷ INTERSEZIONE: (insieme, insieme)  $\longrightarrow$  insieme

$$\text{INTERSEZIONE}(A, B) = C$$

$$\text{Post: } C = A \cap B$$

▷ DIFFERENZA: (insieme, insieme)  $\longrightarrow$  insieme

$$\text{DIFFERENZA}(A, B) = C$$

$$\text{Post: } C = A - B$$

Gli operatori di **INSERISCI** e **CANCELLA** sono ridondanti in quanto casi particolare di **UNIONE** e **DIFFERENZA** in cui il primo argomento é un insieme di un solo elemento.

Sono stati definiti poiché sono molto usati e compaiono in tutte le specifiche degli insiemi, ovvero sono stati definiti per comodità.

## Realizzazione di Insiemi con Vettore Booleano

Supponiamo che gli elementi degli insiemi siano numeri interi compresi tra 0 ed  $N - 1$ .

Allora posso rappresentare un insieme  $A$  tramite un vettore booleano  $I$  di dimensione  $N$ , con la seguente convenzione:

- ▷  $\forall k \in [0 \dots N - 1], I[k] = 1$  se  $k \in A$
- ▷  $\forall k \in [0 \dots N - 1], I[k] = 0$  se  $k \notin A$

In C, al fine di risparmiare memoria, un vettore di valori booleani può essere rappresentato tramite un vettore di elementi di tipo **char**.

Utilizzando questa implementazione é molto semplice implementare le operazioni di:

- ▷ appartenenza: `if (I[k]) ...`
- ▷ inserzione: `I[k] = 1`
- ▷ cancellazione: `I[k] = 0`

infatti, basta accedere direttamente al  $k$ -esimo elemento del vettore ed eseguire una opportuna operazione.

Le operazioni di **unione**, **intersezione** e **differenza** di due insiemi si realizzano tramite **operazioni logiche** sugli elementi del vettore che rappresentano gli insiemi.

Infatti:

- ▷ l'unione si implementa tramite una operazione di OR: **A or B**
- ▷ l'intersezione si implementa tramite una operazione di AND: **A and B**
- ▷ la differenza, cioè tutti quelli del primo insieme che non appartengono al secondo insieme, si implementa tramite un operazione di AND negando i bit del secondo vettore: **A and (not B)**

## Realizzazione di Insiemi con Vettore Booleano

```
#define N 1024 // numero massimo degli elementi dell'insieme
```

```
typedef int    tipoelem;  
typedef char[] insieme;  
typedef char   boolean;
```

```
insieme CREAinsieme ( ) {  
    int i;  
    insieme A;  
    A = (insieme) malloc (N * sizeof(boolean));  
    for ( i = 0; i < N; i++ )  
        A[i] = 0;  
}
```

```
boolean INSIEMEvuoto ( insieme A ) {  
    int i = 0;  
    boolean vuoto = 1;  
    while ( vuoto && ( i < N ) ) { // ← why while instead of for ??  
        vuoto = (A[i] == 0) ? 1 : 0;  
        i++;  
    }  
    return vuoto;  
}
```

```
boolean APPARTIENE ( tipoelem e, insieme A ) {  
    return ( A[e] );  
}
```

```
void INSERISCI ( tipoelem e, insieme A ) {  
    A[e] = 1;  
}
```

```
void CANCELLA ( tipoelem e, insieme A ) {  
    A[e] = 0;  
}
```

```
void UNIONE ( insieme A, insieme B, insieme C ) {  
    int i;  
    for ( i = 0; i < N; i++ )  
        C[i] = A[i] || B[i];  
}
```

```
void INTERSEZIONE ( insieme A, insieme B, insieme C ) {  
    int i;  
    for ( i = 0; i < N; i++ )  
        C[i] = A[i] && B[i];  
}
```



```
void DIFFERENZA ( insieme A, insieme B, insieme C ) {  
    int i;  
    for ( i = 0; i < N; i++ )  
        C[i] = A[i] && ! B[i];  
}
```

L'implementazione tramite vettore booleano ha i seguenti vantaggi:

- ▷ é semplice
- ▷ APPARTIENE, INSERISCI e CANCELLA sono  $\mathcal{O}(1)$
- ▷ CREAinsieme, INSIEMEvuoto, UNIONE, INTERSEZIONE e DIFFERENZA sono  $\Theta(N)$  perché occorre scandire tutto il vettore, ed è il meglio che posso fare

L'implementazione tramite vettore booleano presenta due svantaggi:

- ▷ spreca memoria: nel caso C alloca un char per ogni elemento dell'insieme. Ogni insieme richiede uno spazio  $\Theta(N)$
- ▷ non é espandibile: se l'insieme cresce bisogna riallocare un nuovo vettore
- ▷ alcune operazioni sono troppo lente poiché richiedono  $\mathcal{O}(N)$  anche su insiemi di pochissimi elementi

**Esercizio:** definire una implementazione degli insiemi tramite vettore di interi, in cui ogni elemento rappresenta 32 elementi dell'insieme.

## Realizzazione di Insiemi tramite Lista non Ordinata

Si può rappresentare un **insieme** con una lista i cui elementi sono quelli dell'insieme.

Utilizzando questa implementazione si ha il duplice vantaggio che:

- ▷ gli elementi dell'insieme non devono necessariamente essere compresi nell'intervallo  $1 \dots N$
- ▷ l'occupazione di memoria cresce linearmente con il numero degli elementi dell'insieme

Le operazioni sull'insieme si riducono quindi a operazioni di scansione, inserzione e cancellazione sulle liste.

- ▷ le operazioni di **CREAINSIEME** e **INSIEMEVUOTO** sono  $\mathcal{O}(1)$
- ▷ le operazioni di **APPARTIENE**, **INSERISCI** e **CANCELLA** sono  $\mathcal{O}(n)$  ove  $n$  é la cardinalità dell'insieme.

La complessità dell'operazione **INSERISCI** é determinata dal fatto che prima di inserire un elemento bisogna verificare che tale elemento non esista.

La realizzazione delle operazioni di **UNIONE**, **INTERSEZIONE** e **DIFFERENZA** sono molto simili tra di loro.

Siano  $A$ ,  $B$  gli insiemi di input rispettivamente di cardinalità  $n$  ed  $m$ , e  $C$  l'insieme di output

▷ operazione di **UNIONE**:

1. si copiano gli elementi di  $A$  in  $C$ .
2. per ogni elemento  $b \in B$  se  $b \notin C$  si inserisce  $b$  in  $C$

Il costo totale dell'operazione é quindi  $\mathcal{O}(n) + \mathcal{O}(mn) = \mathcal{O}(mn)$

▷ operazione di **INTERSEZIONE**:

1. per ogni elemento  $a \in A$
2. se  $a \in B$ , si inserisce  $a$  in  $C$

Il costo totale dell'operazione é quindi  $\mathcal{O}(nm)$

▷ operazione di **DIFFERENZA**:

1. per ogni elemento  $a \in A$
2. se  $a \notin B$ , si inserisce  $a$  in  $C$

Il costo totale dell'operazione é quindi  $\mathcal{O}(nm)$

Se  $m \in \mathcal{O}(n)$ , allora il costo computazionale, o complessità, delle precedenti operazioni é  $\mathcal{O}(n^2)$ .

## Realizzazione di Insiemi tramite Lista NON Ordinata

Realizziamo ad esempio l'operazione di **INTERSEZIONE**:

```
void INTERSEZIONE ( insieme A, insieme B, insieme C ) {  
    posizione p, q, r;  
  
    C = CREAinsieme();  
  
    p = PRIMOLISTA(A);  
    r = PRIMOLISTA(C);  
  
    while ( ! FINELISTA(p, A) )  
  
        if ( APPARTIENE ( LEGGILISTA(p), B) )  
            INSLISTA ( LEGGILISTA(p), &r );  
  
    p = SUCCLISTA(p);  
  
}
```

- ▷ si scorre l'insieme  $A$
- ▷ per ogni elemento  $a \in A$  si **verifica se appartiene**  $a \in B$ . Se  $a \in B$  si inserisce  $a$  in  $C$
- ▷ si itera fino ad esaurire la scansione dell'insieme  $A$

## Realizzazione di Insiemi tramite Lista Ordinata

Se sugli elementi dell'insieme é definita una relazione di ordinamento totale  $\leq$  ( $\geq$ ), allora un insieme può essere rappresentato con una lista ordina, ad esempio per valori crescenti.

In questo modo la complessità delle operazioni:

- ▷ **CREAINSIEME** ed **INSIEMEVUOTO** rimane  $\mathcal{O}(1)$  poiché si realizzano tramite le corrispondenti operazioni sulle liste.
- ▷ **INSERISCI**, **APPARTIENE** e **CANCELLA** rimane  $\mathcal{O}(n)$ , in quanto bisogna scandire tutta la lista.
- ▷ **UNIONE**, **INTERSEZIONE** e **DIFFERENZA**:

per queste operazioni, la complessità si abbassa a  $\mathcal{O}(n + m)$ . Se  $m \in \mathcal{O}(n)$  allora si ottiene una complessità  $\mathcal{O}(n)$ .

Il miglioramento di complessità delle ultime tre operazioni é dovuto al fatto che nella loro implementazione possiamo evitare di usare l'operazione di **APPARTIENE**.

Infatti, sfruttando l'ordinamento degli elementi degli insiemi, dal confronto di due elementi possiamo capire se un elemento appartiene o meno ad un insieme.

Ad esempio, consideriamo la realizzazione dell'operazione di **DIFFERENZA**:  $C = A - B$ .

Scandisco gli insiemi  $A$  e  $B$  con due cursori, rispettivamente  $p$  e  $q$ .

Siano  $a$  e  $b$  gli elementi riferiti rispettivamente da  $p$  e  $q$ :

1. se  $a < b$ , allora  $a \notin B$  e quindi, inserisco  $a$  in  $C$  e avanzo il cursore  $p$
2. se  $a = b$  allora  $a \in B$  e quindi avanzo entrambi i cursori  $p$  e  $q$
3. se  $a > b$  ( $b \notin A$  ma irrilevante per il calcolo della differenza) avanzo solamente il cursore  $q$

Il confronto termina quando si raggiunge la fine di una delle due liste. Se si è raggiunta la fine della lista  $B$  allora bisogna copiare i rimanenti elementi di  $A$  in  $C$ .

Così facendo ogni elemento di  $A$  e  $B$  viene considerato una sola volta, e quindi la complessità dell'operazione è  $\mathcal{O}(n + m)$ , supponendo che le liste siano realizzate tramite puntatori.

Le stesse considerazioni di complessità possono essere fatte per le operazioni di **UNIONE** ed **INTERSEZIONE**, quindi nella realizzazione degli insiemi tramite liste ordinate tali operazioni hanno complessità **ottima** poiché considerano una sola volta gli elementi degli insiemi su cui operano.

## Implementazione C dell'operazione di INTERSEZIONE:

```
void INTERSEZIONE ( insieme A, insieme B, insieme C ) {  
    posizione p, q, r;  
  
    C = CREAinsieme();  
    p = PRIMOLISTA(A);  
    q = PRIMOLISTA(B);  
    r = PRIMOLISTA(C);  
  
    while ( ! FINELISTA(p, A) && ! FINELISTA(q, B) )  
  
        if ( LEGGILISTA(p, A) == LEGGILISTA(q, B) ) {  
            INSLISTA ( LEGGILISTA(p, A), &r );  
            p = SUCCLISTA(p);  
            q = SUCCLISTA(q);  
            r = SUCCLISTA(r);  
        } else {  
            if ( LEGGILISTA(p, A) < LEGGILISTA(q, B) )  
                p = SUCCLISTA(p);           // elem puntato da p non esiste in B  
            else  
                q = SUCCLISTA(q);           // elem puntato da q non esiste in A  
        }  
    }  
}
```

L'operazione effettua una unica scansione degli insiemi, quindi  $\text{INTERSEZIONE} \in \mathcal{O}(n + m)$ .

L'implementazione degli insiemi tramite liste ordinate ha inoltre il seguente vantaggio.

Consideriamo le seguenti due operazioni:

▷ MIN: insieme  $\longrightarrow$  tipoelem

$\text{MIN}(A) = x$

Pre:  $A \neq \phi$

post:  $x \in A, \forall y \in A, x \leq y$

▷ MAX: insieme  $\longrightarrow$  tipoelem

$\text{MAX}(A) = x$

Pre:  $A \neq \phi$

post:  $x \in A, \forall y \in A, x \geq y$

In una implementazione qualsiasi la loro realizzazione ha una complessità  $\mathcal{O}(n)$  con  $n$  numero degli elementi dell'insieme.

L'implementazione degli insiemi tramite liste ordinate permette invece di calcolare il **minimo** di un insieme in  $\mathcal{O}(1)$ , basta accedere al primo elemento della lista.

Se le liste utilizzate sono di tipo bidirezionale circolare allora anche in calcolo del **massimo** é  $\mathcal{O}(1)$ .



## MFSET: Merge-Find-Set

Un'altra struttura dati che permette di rappresentare in modo efficiente insiemi disgiunti è quella che va sotto il nome di **Merge-Find-Set** o **MFSET**.

**MFSET** rappresenta una partizione di un insieme finito in sottoinsiemi disgiunti chiamati **componenti** o **parti**.

Le operazioni definite sulla struttura dati MFSET sono:

- ▷ **find**: permette di stabilire a quale componente appartiene un generico elemento
- ▷ **merge**: unisce due componenti distinte in un sola componente, distruggendo le due vecchie componenti, lasciando inalterate le altri componenti

Rispetto al classico tipo di dato insieme non sono previste operazioni di cancellazione o inserzione di elementi fatta eccezione per l'operatore di **CREAMFSET**.

- ▷ l'operazione **CREAMFSET** crea un **MFSET** composto da una partizione in cui ogni componente contiene un solo elemento
- ▷ l'operazione **TROVA** restituisce l'identificatore di una componenente
- ▷ l'operazione **FONDI** unisce due componenti senza modificare le altre e dopo tale esecuzione il numero di componenti é diminuito di uno

## MFSET: Specifica

La specifica dell'**MFSET** é la seguente:

▷ **CREAMFSET**: (insieme)  $\longrightarrow$  mfset  
 $\text{CREAMFSET}(A) = S$

Post:  $S$  é una famiglia di  $n = |A|$  componenti  $C_1, \dots, C_n$  ciascuna delle quali contiene un solo elemento e tali che  $\bigcup_{1 \leq i \leq n} C_i = A$

▷ **TROVA**: (tipoelem, mfset)  $\longrightarrow$  componente  
 $\text{TROVA}(x, S) = C$

Pre:  $\exists j \mid x \in C_j$

Post:  $C$  é l'identificatore della componente a cui appartiene  $x$

▷ **FONDI**: (tipoelem, tipoelem, mfset)  $\longrightarrow$  mfset  
 $\text{FONDI}(x, y, S) = S'$

Pre:  $x \in C_i, y \in C_j, i \neq j$ , cioè  $x, y$  appartengono a due componenti diverse

Post:  $S'$  contiene tutte le componenti di  $S$  che non contengono  $x$  ed  $y$ , piú una nuova componente data dall'unione delle due componenti contenenti  $x$  e  $y$ . Insiemeisticamente possiamo scrivere:

$$S' = S - C_i - C_j \cup C_k, \quad C_k = C_i \cup C_j$$

## MFSET: Realizzazione

Una realizzazione efficiente della struttura dati **MFSET** si ottiene memorizzando l'**MFSET** in un albero **NON** ordinato o **foresta**.

Ogni componente dell'**MFSET** é identificata con la radice dell'albero corrispondente.

Utilizzando questa rappresentazione la realizzazione dell'operazione di:

- ▷ **TROVA** deve restituire la radice dell'albero corrispondente alla componente a cui appartiene l'elemento  $x$
- ▷ **FONDI** unisce due alberi in uno.

Se é possibile accedere direttamente ai nodi contenenti gli elementi allora la realizzazione dell'operazione di:

- ▷ **TROVA** deve risalire, attraverso l'operazione di **PADRE**, alla radice dell'albero
- ▷ **FONDI** opera in modo analogo all'operazione di **TROVA** per risalire alle radice degli alberi delle due componenti da unire e poi una delle due radice diviene figlio dell'altra

Poiché entrambe le operazioni di **TROVA** e **FONDI** prevedono di risalire da un nodo alla radice, la **complessità** delle due operazioni é **lineare** nell'altezza degli alberi corrispondenti alle componenti su cui operano.

Al fine di mantenere basso il livello degli alberi che rappresentano le componenti, l'operazione di **FONDI** deve essere implementata in modo tale che l'albero ottenuto come risultato della fusione sia il più possibile **bilanciato**.

Per ottenere tale risultato l'operazione di **FONDI** adotta la seguente politica di unione di due componenti:

- ▷ si sceglie come radice del nuovo albero quella della componente con il maggior numero di nodi.
- ▷ la radice dell'albero della componente con il numero minore di nodi diviene un nuovo figlio della radice dell'altra componente.

Adottando tale politica qual'è nel **caso pessimo** l'altezza dell'albero che si ottiene applicando una successione di operazione di **FONDI** ad un insieme iniziale di  $n$  componenti ciascuno di cardinalità 1 ?

Consideriamo un elemento  $x$  inizialmente appartenente ad una delle  $n$  componenti,  $C_x$ , di cardinalità 1. Inizialmente il livello, distanza dalla radice, di  $x$  è  $l(x) = 0$  e ci chiediamo di quanto può crescere ad ogni operazione di fusione.

Il livello  $l(x)$ :

- ▷ sarà invariato se  $x$  appartiene alla componente di cardinalità maggiore
- ▷ aumenterà di uno se  $x$  appartiene alla componente di cardinalità minore.

Il caso pessimo si ottiene, quindi, facendo in modo che  $x$  appartenga sempre alla componente di cardinalità minore, ovvero procedendo secondo il seguente algoritmo:

- ▷ sia  $C_x$  la componente a cui appartiene  $x$
- ▷ si costruisca una componente  $C_k$  tale che  $|C_k| = |C_x|$
- ▷ si uniscano le componenti  $C_k$  e  $C_x$  in modo tale che la nuova radice sia la radice della componente  $C_k$
- ▷ si rinomini  $C_x$  la nuova componente ottenuta e si riapplichino l'algoritmo

Così facendo, ad ogni operazione di fusione il numero dei nodi dell'albero ottenuto raddoppia, ovvero il numero delle componenti dell'insieme iniziale si dimezza, quindi al più posso fare  $\log n$  operazioni di fusioni.

In tal modo, il livello di  $x$  sarà al più  $\mathcal{O}(\log n)$ .

Da ciò si deduce che le operazioni di **TROVA** e **FONDI** operano su alberi la cui altezza  $h \in \mathcal{O}(\log n)$  ove  $n$  è il numero dei nodi dell'albero cioè il numero degli elementi dell'insieme.

Quindi la complessità delle operazioni **TROVA** e **FONDI** è  $\mathcal{O}(\log n)$ .

La complessità dell'operazione di **CREAMFSET** invece è  $\Theta(n)$ , ove  $n$  è il numero degli elementi dell'insieme.

## MFSET: Realizzazione

Supponiamo, per semplicità, che gli elementi dell'insieme  $A$  siano tutti i numeri interi compresi nell'intervallo  $[0 \dots n - 1]$ , e l'insieme  $A$  sia rappresentato con una variabile intera contenente il valore  $n$ .

La foresta che rappresenta l'**MFSET** può essere rappresentato tramite due vettori:

- ▷ il vettore dei padri per rappresentare le componenti, cioè gli alberi corrispondenti alle componenti dell'insieme
- ▷ il vettore delle **dimensioni** che per ogni radice contiene il numero dei nodi dell'albero

```
typedef int tipoelem;  
  
typedef int componente;  
  
typedef int insieme;  
  
struct mfset_struct {  
    int padre;  
    int dimensione;  
};  
  
typedef mfset_struct mfset[MAXLUNG];
```

La procedura **CREAMFSET** inizializza la foresta con  $n$  alberi di un solo elemento:

```
void CREAMFSET ( insieme A, mfset S ) {  
    int i;  
  
    for ( i=0; i < A; i++ ) {  
        S[i].padre      = -1;  
        S[i].dimensione = 1;  
    }  
}
```

La funzione **TROVA** dato un elemento risale alla radice che viene restituita come identificatore della componente.

```
componente TROVA ( tipoelem x, mfset S ) {  
  
    if ( S[x].padre == -1 )  
        return x;  
    else  
        return TROVA( S[x].padre , S );  
}
```

La procedura **FONDI** richiama due volte la procedura TROVA per ritrovare le radici dei due alberi, e poi utilizza il vettore dimensione per stabilire qual'è l'albero più piccolo.

```
void FONDI (tipoelem x, tipelem y, mfset S) {
    componente i, j;

    i = TROVA(x, S);
    j = TROVA(y, S);

    if ( i != j )

        if ( S[i].dimensione < S[j].dimensione ) {
            S[i].padre = j;
            S[j].dimensione += S[i].dimensione;
        } else {
            S[j].padre = i;
            S[i].dimensione += S[j].dimensione;
        }
}
```

Come precedentemente dimostrato:

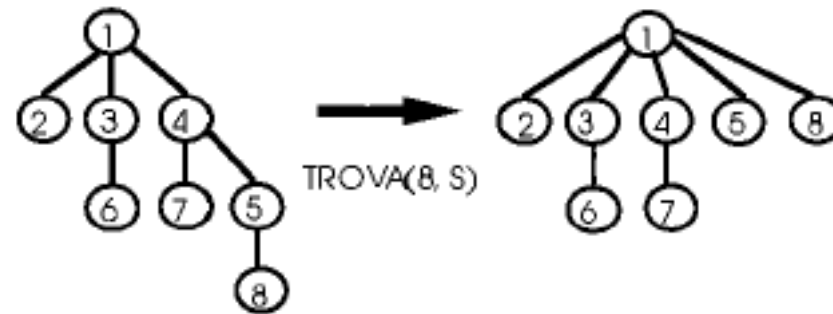
- ▷ **TROVA** e **FONDI** hanno complessità  $\mathcal{O}(\log n)$
- ▷ **CREAMFSET** ha complessità  $\Theta(n)$



## MFSET: Compressione dei percorsi

Per tenere bassa la complessità media delle operazioni di **TROVA** e **FONDI** si ricorre alla tecnica della **compressione dei percorsi**.

La tecnica consiste nel rendere figlio della radice ogni nodo che viene incontrato dall'operazione di **TROVA** nel percorso di risalita da un nodo alla radice.



```
componente TROVA ( tipoelem x, mfset S ) {
    if ( S[x].padre == -1 )
        return x;
    else {
        S[x].padre = TROVA( S[x].padre, S );
        return S[x].padre;
    }
}
```

Il vantaggio introdotto dalla compressione dei percorsi si ha quando viene effettuata la sequenza di operazioni:

- ▷ CREAMFSET
- ▷  $m$  FONDI,  $m \geq n$  ( $n$  numero di componenti)
- ▷  $m$  TROVA,  $m \geq n$  ( $n$  numero di componenti)

Senza la compressione dei percorsi la sequenza ha complessità:

$$\mathcal{O}(n + m \cdot \log n)$$

Con la compressione dei percorsi é possibile dimostrare che il costo diviene

$$\mathcal{O}(n + m\alpha(m, n))$$

ove  $\alpha$  é la funzione inversa della funzione di Ackermann

La funzione di Ackermann é una funzione **iper-esponenziale** quindi la sua inversa cresce molto lentamente, tanto che  $\alpha(m, n) \leq 4$  per ogni valore di  $m, n$  rappresentabile su 32 bit.

Conseguenza di cioé é che nel caso medio le operazioni di TROVA e FONDI hanno costo quasi  $\mathcal{O}(1)$  anche per valori di  $m$  ed  $n$  molto grandi.

## Esercizi

Scrivere un algoritmo per il calcolo di ognuno dei seguenti operatori:

- ▷ intersezione, unione, differenza e differenza simmetrica di insiemi rappresentati tramite liste non ordinate
- ▷ intersezione, unione, differenza e differenza simmetrica di insiemi rappresentati tramite liste ordinate

**N.B.:** la differenza simmetrica tra due insiemi  $A$  e  $B$  é si calcola come

$$A \triangle B = (A \cup B) - (A \cap B)$$