**MAHNATTAN**

(define 2D ; valore: naturale

(lambda (down right) ; down, right: naturali

(if (or (= down 0) (= right 0))

;se una delle 2 coordinate è 0, si ha 1 strada

;rettilinea fino alla destinazione

1

(+ (2D (- down 1) right) ;procede verso il BASSO

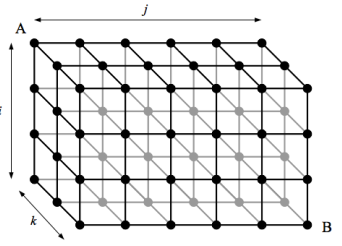
(2D down (- right 1)) ;procede verso DESTRA

)

)

)

)



(define manhattan-3d

(lambda (down right k) ;k(3°dimensione), down, right: numeri interi

(cond

;se almeno 1 dimensione è 0 si passa alla valutazione BIDIMENSIONALE

((= down 0)

(2D right k)

)

((= right 0)

(2D down k)

)

((= k 0)

(2D down right)

)

(else

(+

(manhattan-3d (- down 1) right k) ;procede verso il BASSO

(manhattan-3d down (- right 1) k) ;procede verso DESTRA

(manhattan-3d down right (- k 1)) ;procede verso la 3°dimensione

)

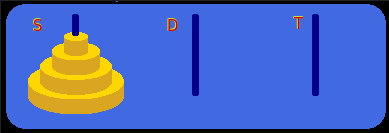
)

)

)

)

**HANOI**

****

(define hanoi-moves

(lambda (n) ; **N: numero dei dischi**

(hanoi-rec n 1 2 3) ;1 2 3 sono gli indici che individuano le aste **S D T**

)

)

;**NUMERO TOTALE DI MOSSE = 2^N**

;ritorna la **lista totale di mosse dalla prima all'ultima**

(define hanoi-rec ;ritorna la lista totale di mosse dalla prima all'ultima

(lambda (n s d t) ;source destination transit; n = numero dischi

(if (= n 1) ;rimasto 1 disco S --> D

(list (list s d)) ;MOSSA S --> D

;MOSSA 1 = sposta i primi **n-1 dischi da S --> T**

;MOSSA 2 = poi sposta **nesimo disco da S --> D**

;MOSSA 3 = per ultimo sposta **n-1 dischi da T --> D**

(let ( (m1 (hanoi-rec (- n 1) s t d) ) **;source = S destination = T**

(m2 (list s d)) ;**ultimo disco da S --> D**

(m3 (hanoi-rec (- n 1) t d s)); **source = T, destination = D**

)

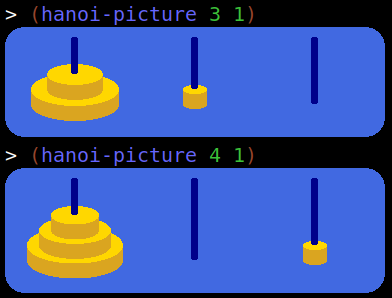
;; costruisce la lista delle **mosse in ordine da m1 a m3**

(append

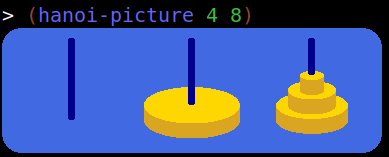
m1

(cons m2 m3)

) ))))



1. Configurazione dopo la **mossa 1 di hanoi-moves 3**
2. Condigurazione dopo la **mossa 1 di hanoi-moves 4**
3. Mossa 1 (hanoi-moves 4) == (hanoi-rec (- n 1) s t d) == **Mossa 1 (hanoi-rec 3)**
4. La **destinazione** del **sottoproblema** = **Mossa 1 (hanoi-moves 4) è T,**
   1. per spostare il blocco(n) più grande, tutti gli altri(n-1) devono essere spostati in T
   2. D ora è libero per poter contenere n-esimo blocco = MOSSA 2
5. **MOSSA 3 di (hanoi-moves 3): (n-1)blocchi** **da T** devono essere **spostati in D**
   1. La **destinazione** rimane la **stessa = D**, mentre il **punto** di **partenza** è **T**, invece di S



Alla **mossa intermedia[2^(n-1)] di (hanoi-moves n)**  si ha questa situazione…

**Perchè 2^(n-1) ???**

**Quante mosse** servono per **spostare (n-1)blocchi da S a T?**

**= 2^(n-1)**

;; Restituisce la **configurazione al termine della "k"-esima mossa**

(**define hanoi-disks** ; hanoi-disks: lista delle asticelle con il relativo numero di blocchi piazzati

(lambda (n k) ; **n = numero dischi**, **k = mossa inclusa** a cui **terminare** la **ricerca**

(**hanoi-rec-cod** n k

;configurazioni iniziali delle **3 postazioni con 0 dischi**

'(1 0) ;SOURCE

'(2 0) ;DESTINATION

'(3 0) ;TRANSIT

n) ;**n backup**

)

)

;ricorsione di CODA: si stabilisce la posizione dell n-esimo disco, quindi il maggiore e via via sia arriva al minore

**(define hanoi-rec-cod**  ; hanoi-rec-cod: list

(lambda (n k s d t n\_backup) ; n, k, n\_backup: int ; s, d, t: lists

(let ( **(l (expt 2 (- n 1)))** ;**mossa intermedia** = 2^(n-1)

**(h (+ (cadr s) (cadr d) (cadr t)))** ;somma i secondi elementi delle liste

;**somma** dei **contatori** che indicano i **pezzi** **piazzati**

)

(cond

(**(= h n\_backup)**  ;**mosse terminate** perchè h è la somma di **tutti i pezzi spostati**

(list s d t)

)

(**(< k l)**  ;mosse **precedenti** a quella intermedia

(hanoi-rec-cod (- n 1) k

(list (car s) (+ (cadr s) 1)) ;i**l blocco n-esimo si troverà ancora nella S**

t d n\_backup) **;trasferisce i n-1 blocchi da S a T passando per D**

)

(**(not (< k l))** ;mossa successiva a quella intermedia

(hanoi-rec-cod (- n 1)

(- k l) ;n-esimo blocco piazzato

t ;n-1 blocchi da T a D passando per S

(list (car d) (+ (cadr d) 1)) ;incrementa il numero di oggetti in D

s n\_backup)

)

)

)

)

)

**;ricorsione di CODA: si stabilisce la posizione dell n-esimo disco,**

; **e via via sia arriva al minore**

**(define hanoi-rec-cod**  ; hanoi-rec-cod: list

(lambda (n k s d t n\_backup) ; n, k, n\_backup: int ; s, d, t: lists

(let ( (l (expt 2 (- n 1))) ;lunghezza = 2^(n-1)

(h (+ (cadr s) (cadr d) (cadr t))) ;**somma i contatori delle posizioni**

)

(cond

((= h n\_backup)  **;mosse terminate** perchè h è la somma di tutti i pezzi spostati

(list s d t)

)

((< k l) **;mosse precedenti a quella intermedia**

;**deve ancora trasferire tutti gli (n-1)blocchi da S a T = mossa 1**

(hanoi-rec-cod (- n 1) k

(list (car s) (+ (cadr s) 1)) **;il blocco n-esimo si troverà ancora nella S**

t d n\_backup)

)

((not (< k l)) **;mossa successiva a quella intermedia**

(hanoi-rec-cod (- n 1)

(- k l) **;toglie la mossa intermedia**

t **;n-1 blocchi da T a D passando per S = mossa3**

(list (car d) (+ (cadr d) 1)) **;incrementa il numero di oggetti in D = mossa 2**

s

n\_backup)

)

)

)

)

)

**Restituisce l’immagine della configurazione dopo k mosse di (hanoi-moves3)**

(define recursive-picture

(lambda (N n k s d t ns nd nt img) ;**N: numero torale dei dischi da usare** per la **dimensione delle figure tramite il teachpack**

;**n: dischi ancora da spostare,** progressivamente diminuisce fino a 0, quando tutti saranno stati piazzati

**;k: mosse da trovare**

;**s d t = posizioni; ns nd nt: numero di disci per ogni posizione**

;image : immagine generata in precedenza a cui sovrapporre la nuova

(if (= n 0)

img ;**tutti I dischi sono stati piazzati**

(if (< k (expt 2 (- n 1)))

**;mossa precedente a quella intermedia**

;**deve ancora finire di spostare (n-1)blocchi da S a T,**

;oppure ha appena finito ma non si può sapere

(recursive-picture N (- n 1) k **;configurazione di (n-1) blocchi dopo la k-mossa**

s t d ;da S --> T

(+ ns 1) **;il n-esimo si trova in S**

nt nd

(above (disk-image (- n 1) N s ns) img) ;nuova immagine generata

**;viene piazzato l'n-esimo blocco nell'asticella S**

)

**;mossa successiva o uguale**

;**n-esimo è sicuramente stato spostato in D**

;--> **si procede con le mosse in ordine giusto**

;=**= (n-1)blocchi da T a D**

(recursive-picture N (- n 1) (- k (expt 2 (- n 1)))

t d s ;da T --> D

nt

(+ nd 1) ;n-esimo si trova in D

ns

(above (disk-image (- n 1) N d nd) img) ;nuova immagine generata

)

)

)

)

)

;restituisce la configurazione grafica della situazione dopo K mosse

(define **hanoi-picture**

(lambda (n k)

(recursive-picture n n k

1 2 3 ;; S D T

0 0 0 ;ns nd nt

(towers-background n) ;imposta la dimensione dello sfondo

)

)

)

**LCS**

; LCS = LONGEST COMMON SUBSEQUENCE

(define llcs ;determina la lunghezza (INTERO) della LCS

(lambda (a b) ;a b = STRINGHE

(cond

((or (string=? a "") (string=? b "") ) ;se una delle 2 stringhe è vuota

0 ;la LCS di "" "testoCASUALE" è "", perciò la sua lunghezza è 0

)

((char=? (string-ref a 0) (string-ref b 0))

;se le due stringhe hanno lo stesso primo carattere

(+ 1 (llcs (substring a 1) (substring b 1)) )

)

(else ;le due stringhe iniziano con 2 char diversi

(max

(llcs (substring a 1) b)

(llcs a (substring b 1))

)

)

)

)

)

(define longer ;restituisce la stringa più lunga tra le 2

(lambda (str1 str2) ;STRINGA

(let (

(k1 (string-length str1))

(k2 (string-length str2))

)

(if (< k1 k2)

str2 ;str1 < str2

str1 ;str1 >= str2

)

)

)

)

;esempio "13242132" "2423142" stringhe causali di 4 simboli

;i simboli delle stringhe possono essere anche diversi

;LCS determina la sottosequenza comune ordinata da SX a DX delle due stringhe inserite

(define lcs ;restituisce la STRINGA LCS

(lambda (a b) ;a b = STRINGHE

(cond

((or (string=? a "") (string=? b ""))

"" ;se una stringa è vuota si restituisce LCS = ""

)

((or (= (string-length a) 1) (= (string-length b) 1))

;in caso di stringa lunga 1 solo char controllare il char con tutti gli altri della stringa maggiore

(if (= (string-length a) 1)

(if (char=? (string-ref a 0) (string-ref b 0)) ;controlla il primo carattere delle due stringhe

(string-append

a

(lcs a (substring b 1)) ;toglie il primo carattere di b

)

(lcs a (substring b 1))

)

(if (char=? (string-ref a 0) (string-ref b 0)) ;controlla il primo carattere delle due stringhe

(string-append

b

(lcs (substring a 1) b) ;toglie il primo carattere di a

)

(lcs (substring a 1) b )

)

)

)

( (char=? (string-ref a 0) (string-ref b 0)) ;le due stringhe iniziano con lo stesso char

; carattere uguale + la LCS delle due stringhe senza il primo carattere

(string-append

(substring a 0 1) ;PRIMO CARATTERE di A o B

(lcs (substring a 1) (substring b 1)) ;toglie il primo carattere da entrambe le stringhe

)

)

(else

;se le due stringhe inziano con char diversi

(longer ;restituisce la stringa più lunga tra due LCS

(lcs (substring a 1) b) ;toglie il primo char di a

(lcs a (substring b 1)) ;toglie il primo char di b

)

)

)

)

)

**LIST-FREQ**

(define list-freq ; val: intero

(lambda (c lst) ; conta le occorrenze di c negli elementi della lista

(proc (map ;argomento procedurale

(str-freq c) ;valore procedurale

lst

)

)

)

)

(define proc ;restituisce la somma degli argomenti numerici nella lista

(lambda (lista)

(if (null? lista) ;0 elementi

0

(+ (car lista) ;primo elemento della lista

(proc (cdr lista))

)

)

)

)

(define str-freq ;restituisce un numero intero di occorrenze del carattere char nella stringa

(lambda (char) ;char da contare

(lambda (stringa) ;valore procedurale

(cond ( (string=? stringa "")

0 ;CASO BASE

)

((char=? (string-ref stringa 0) char)

(+ 1

((str-freq char )(substring stringa 1)) ;toglie il primo carattere

)

)

(else

((str-freq char )(substring stringa 1))

)

)

)

)

)

(define lista-freq

(lambda (n lista) ;conta le occorrenze di n nella lista di numeri

(proc (map

(num-freq n)

lista

)

)

)

)

(define num-freq

(lambda (n)

(lambda (lst) ;valore procedurale

(cond ( (null? lst)

0 ;CASO BASE

)

((= (car lst) n)

(+ 1

((num-freq n )(cdr lst)) ;toglie il primo elemento

)

)

(else

((num-freq n )(cdr lst))

)

)

)

)

)

**RICORSIONE CODA**

(define btr-val-tr ; val: intero

(lambda (btr) ; btr: stringa di – / . / +

(btr-val-rec btr 0 0)

))

(define btr-val-rec

(lambda (btr current index) ;current: intero decimale progressivo della conversione

(let ((k (string-length btr))

)

(if (= k 0)

current

(let ((q (substring btr 0 (- k 1))) ;toglie l'ultimo carattere

(t (string-ref btr (- k 1))) ;ultimo carattere da convertire

)

(btr-val-rec q

(+ current (\* (expt 3 index) (btd-val t)))

(+ 1 index)

)

)

)

)

))

;; Questo file contiene esempi di codice Scheme:

;; Ricorsione di coda e correttezza - 5/11/03

;; Ultimo aggiornamento: 3/12/15

;; Massimo Comun Divisore (MCD)

(define mcd ; valore: naturale

(lambda (x y) ; x, y: naturali positivi

(cond ((= x y) x)

((< x y) (mcd x (- y x)))

(else (mcd (- x y) y)) ; x > y

)

))

;; Algoritmo del contadino Russo per la moltiplicazione:

;; Procedura che applica la ricorsione di coda

(define peasant-muliplication ; valore: naturale

(lambda (m n) ; m, n: naturali

(peasant-mul-tr m n 0)

)) ; valore risultante: mn

(define peasant-mul-tr ; valore: naturale

(lambda (x y z) ; x, y, z: naturali

(cond ((= y 0)

z)

((even? y)

(peasant-mul-tr (\* 2 x) (quotient y 2) z))

(else ; y dispari

(peasant-mul-tr (\* 2 x) (quotient y 2) (+ z x)))

)

)) ; valore risultante: z + xy

;; Dimostra per induzione che:

;;

;; (peasant-mul-tr x y z) -\*-> z + xy

;;

;; Quindi dimostra che:

;;

;; (peasant-muliplication m n) -\*-> mn

;; Quali funzioni di n calcolano

;; le procedure odd e unknown?

(define odd ; valore: ?

(lambda (i) ; i > 0 naturale

(if (= i 1)

1

(+ (odd (- i 1)) 2)

)

))

(define unknown ; valore: ?

(lambda (x) ; x: naturale

(if (= x 0)

0

(+ (unknown (- x 1)) (odd x))

)

))

;; Una volta individuate le formule che rappresentano il valore

;; delle espressioni (odd n) e (unknown n) al variare di n nel

;; rispettivo dominio (specifiche), dimostrane la correttezza.

;; Qual'e' la sequenza dei valori di (ufo n) per n = 1, 2, 3, ...?

;; Che proprieta' ha?

(define ufo ; valore: ?

(lambda (x) ; x > 0 naturale

(cond ((= x 1) 1)

((even? x) ; x pari

(- (\* 2 (ufo (quotient x 2))) 1))

(else ; x dispari

(+ (\* 2 (ufo (quotient x 2))) 1))

)

))

;; Identifica le proprieta' del valore dell'espressione (ufo n)

;; in relazione al valore n del parametro.

;; Quale funzioni di n calcola la procedura mistery?

;; (integrazione del 29/10/04 --- ricorsione di coda)

(define mistery ; valore: ?

(lambda (n) ; n naturale

(process n 1 0)

))

(define process ; valore: ?

(lambda (i x y) ; i, x, y naturali, x > 0

(if (= i 0)

y

(process (- i 1) (+ x 2) (+ y x))

)

))

;; Una volta individuata la formula che rappresenta il valore

;; dell'espressione (mistery n) al variare di n nei naturali

;; (specifiche), dimostra la correttezza di "mistery"

;; riconducendola alla correttezza di "process".

**CRITTOGRAFIA PROCEDURALE**

;LE SEGUENTI FUNZIONI VERRANNO PASSATE a crittazione come chiave di crittazione

(define regola-cesare ;cambiando la procedura, cambia anche la chiave di crittografia e quindi l'interpretazione di un carattere

(lambda (key) ; chiave di crittazione passata durante la chiamata

(lambda (x) ;valore procedurale restituito dalla funzione

(integer->char

(let ( (i (+ key (char->integer x))) ;;cod ASCII di X shiftata di key

)

(if (<= i posZ)

i

(- i 26)

)

)

)

)

)

)

(define crittazione ;resituisce una stringa contente il msg cifrato in base alla funzione espressa da reg

(lambda (msg reg)

(if (string=? msg "")

""

(string-append

(string (reg (string-ref msg 0)) ) ;coverte il primo carattere della stringa msg

(crittazione (substring msg 1) reg)

)

)

)

)

;(crittazione (crittazione "CIAOCIAO" (regola-cesare 3)) testo cifrato

; (regola-cesare 23)

;) RESTITUISCE IL TESTO INIZIALE 23 + 3 = 26 = rotazione completa

**MAP**

(define funzione

(lambda (x)

(+ x 1) ;restituisce il successivo

)

)

(define lista ‘(2 5 7 44 1 102))

(map funzione lista)

‘(3 6 8 45 2 103)