Stirling

- Ricorsione ad albero: Numeri di Stirling del II tipo.
  - Formulazione del problema ed esemplificazione:
    - In quanti modi si possono ripartire n elementi in k gruppi non vuoti? Quante sono le possibili partizioni di un insieme di cardinalita' n in k sottoinsiemi non vuoti?
  - Problema generale.
    - Determinazione del numero di partizioni di un insieme di n elementi in k sottoinsiemi non vouti: in quanti modi e' possibile ripartire…?
    - Approccio ricorsivo - impostazione: fissato un particolare elemento, per formare una partizione posso scegliere se questo stara' da solo o assieme ad altri elementi…
- Impostazione di una soluzione ricorsiva generale.
  - Riduzioni ricorsive ($1 < k < n$), preso un oggetto X:
    - se X costituisce un gruppo da solo restano S(n-1,k-1) possibilita' di ripartire gli altri n-1 oggetti in k-1 gruppi;
    - altrimenti per ciascuna delle S(n-1,k) ripartizioni che posso formare con n-1 oggetti ho k modi di decidere con chi sta X.
  - Casi base e riduzioni ricorsive:
    - se $k = 1$: quanti modi per ripartire gli oggetti?
    - se $k = n$: quanti modi per ripartire gli oggetti?
  - In sintesi, i numeri di Stirling del II tipo sono cosi' definiti:
    - $S(n,1) = S(n,n) = 1$ per $n > 0$
    - $S(n,k) = S(n-1,k-1) + k\, S(n-1,k)$ per $1 < k < n$

```
(define st       ; val: intero
  (lambda (n k) ; 1 <= k <= n interi
    (if (or (= k 1) (= k n))
        1
        (+ (st (- n 1) (- k 1)) (* k (st (- n 1) k)))
        )
    ))
```

LCS

- Lunghezza della sottosequenza comune piu’ lunga (LLCS)
  - llcs(““,v) = llcs(u,”“) = 0
  - llcs(au,av) = 1 + llcs(u,v)
  - llcs(au,bv) = max(llcs(u,bv), llcs(au,v)) se a <> b
- Problema della sottosequenza comune piu’ lunga: soluzioni (LCS).
  - Calcolo della LCS a partire dalla struttura di “llcs” (qui ‘+’ rappresenta la giustapposizione di stringhe):
    * lcs(““,v) = lcs(u,”“) =””
    * lcs(au,av) = a + lcs(u,v)
    * lcs(au,bv) = longer(lcs(u,bv), lcs(au,v)) se a <> b

```
(define llcs
  (lambda (s1 s2)
    (cond
      ((or (= (string-length s1) 0) (= (string-length s2) 0)) 0)
      ((char=? (string-ref s1 0) (string-ref s2 0))
       (+ 1 (llcs (substring s1 1) (substring s2 1)))
       )
      (else
       (max (llcs (substring s1 1) s2) (llcs s1 (substring s2 1)))
       ))
    ))

(define lcs
  (lambda (s1 s2)
    (cond
      ((or (= (string-length s1) 0) (= (string-length s2) 0)) "")
      ((char=? (string-ref s1 0) (string-ref s2 0))
       (string-append (substring s1 0 1) (lcs (substring s1 1) (substring s2 1)))
       )
      (else
       (longer-string (lcs (substring s1 1) s2) (lcs s1 (substring s2 1)))
       ))
    ))

(define longer-string
  (lambda (s1 s2)
    (let ((s1l (string-length s1)) (s2l (string-length s2)))
      (cond
        ((> s1l s2l) s1)
        ((< s1l s2l) s2)
        ((= (random 2) 0) s1)
        (else s2)
        )
      ))
  )
```

Fibonacci

```
(define adulti    ; val: intero (numero coppie)
  (lambda (t)     ; t: intero non negativo
    (if (= t 0)
        1
        (+ (adulti (- t 1)) (cuccioli (- t 1)))
        )
    ))

(define cuccioli
  (lambda (t)
    (if (= t 0)
        0
        (adulti (- t 1))
        )
    ))

(+ (adulti 12) (cuccioli 12))
```

Lista numeri primi

- Raffinamenti del programma (I):
  - Fra i “candidati” divisori di n, e’ sufficiente considerare quelli dispari (ad eccezione di 2);
  - Verifica ricondotta all’esistenza di divisori DISPARI nell’intervallo [3, n-1].
- Raffinamenti del programma (II):
  - Fra i “candidati” divisori di n, e’ sufficiente considerare quelli minori o uguali alla radice di n [sqrt(n)];
  - Se n = pq e p > sqrt(n), allora q < sqrt(n);
  - Verifica ricondotta all’esistenza di divisori dispari nell’intervallo [3, floor(sqrt(n))];
  - floor(x) e’ la parte intera di x.

```
(define primo?   ; val: boolean
  (lambda (n)    ; n >= 2 intero
    (if (even? n)
        (= n 2)
        (not (divisori-in? n 3 (sqrt n)))
        )
    ))

(define divisori-in?   ; val: boolean
  (lambda (n inf sup)  ; n, inf, sup: interi positivi
    (cond
      ((> inf sup) false)
      ((= (remainder n inf) 0) true)
      (else (divisori-in? n (+ inf 2) sup))
      )
    ))

(define lista-primi ; val: lista di interi
  (lambda (a b)         ; a, b: interi >= 2
    (cond
      ((> a b) null)
      ((primo? a) (cons a (lista-primi (+ a 1) b)))
      (else (lista-primi (+ a 1) b))
      )
    ))

(lista-primi 2 50)
```

Algoritmo del “contadino russo” per la moltiplicazione.

```
(define mul       ; val: intero
  (lambda (m n)   ; m, n: interi non negativi
    (mul-rec m n 0)
    ))

(define mul-rec     ; val: intero
  (lambda (m n p)   ; m, n, p: interi non negativi
    (cond ((= n 0) p)
          ((even? n) (mul-rec (* 2 m) (quotient n 2) p))
          (else ; n dispari
           (mul-rec (* 2 m) (quotient n 2) (+ m p))
           )
          )
    ))
```

MCD

Algoritmo di Euclide per il Massimo Comun Divisore (MCD).

- Proprieta' catturate dalla definizione ricorsiva:
  - MCD(x, x) = x
  - MCD(x, y) = MCD(x, y-x) se x < y
  - MCD(x, y) = MCD(x-y, y) se x > y

  oppure in termini di resto della divisione

  - MCD(x, y) = y se x mod y = 0
  - MCD(x, y) = MCD(y, x mod y) altrimenti
  - [x mod y : resto della divisione intera di x per y]

```
(define mcd       ; val: intero
  (lambda (x y)   ; x, y: interi positivi
    (cond ((= x y) x)
          ((< x y) (mcd x (- y x)))
          (else (mcd (- x y) y))
          )
    ))
```

Commands

Special Forms

(lambda (s1 . . .) expr 1 expr 2 . . .)
Creates a new procedure whose formal parameters are s1, s2, . . .
and whose body is the given list of expressions.

(if test expr 1 expr 2)
Evaluates test, which is an arbitrary Scheme expression.
If the resulting value is #f, expr 2 is evaluated; *otherwise expr 1 is evaluated.*

(cond (test1 elst1) (test2 elst2) . . . [(else elst e)])
Evaluates each test expression in left-to-right order.
If test1 evaluates to a true value, then the corresponding sequence of expressions elsti is evaluated,
and the value of the last expression is returned.
The optional else clause is evaluated if none of the preceding tests yields a true value.

(and expr . . .)
Evaluates expressions from left to right, and returns the value of the first expression that returns #f.
If no expression returns #f, the value of the last expression is returned,
or #t if there are no expressions.

(or expr . . .)
Evaluates expressions from left to right, and returns the value of the first expression that returns
a true value (i.e., not #f). If there are no expressions, or all expressions return #f,
then the or returns #f.

(let ((s1 texpr 1) (s2 texpr 2) . . .) expr 1 expr 2 . . .)
Evaluate expr 1 , expr 2, . . . in an environment with s1 bound to the value of texpr 1 ,
s2 bound to the value of texpr 2, etc.

Standard Procedures

Equality Testing

(= n1 n2 . . .)
Returns #t if n1 = n2 = ... Note that this works only for numbers

List Structure Operations

(null? x) Returns #t if x is the empty list '(), #f for any other Scheme object.
(cons x y) Creates a new pair (of type <pair>) whose car is x and whose cdr is y.
(car c) Returns the first element of c.
(cdr c) Returns the second element of c.
(list expr 1 expr 2 . . .) Create a new list consisting of the values of the given expressions.
(length lst) Return the number of elements in the given list.
(append lst1 lst2) Append the two lists together, and return the resulting list.
(reverse lst) Return a new list which has same elements as lst, but in the opposite order.

Arithmetic and Numeric Operators

(= n1 n2 . . .) (< n1 n2 . . .) (> n1 n2 . . .) (<= n1 n2 . . .) (>= n1 n2 . . .)
Tests whether a sequence of numerical values are, respectively, equal, strictly increasing,
strictly decreasing, nondecreasing, or non-increasing.
(+ n1 n2 . . .) Addition.
(- n1 n2 . . .) Subtraction (negation, with a single argument).
(* n1 n2 . . .) Multiplication.
(/ n1 n2 . . .) Real or rational division (associates to the right).
(quotient n1 n2 . . .) Quotient from integer division
(remainder n1 n2 . . .) Remainder from integer division
(modulo n1 n2 . . .) Least non-negative residue of remainder.
(abs n) Absolute value.
(min n1 n2 . . .) (max n1 n2 . . .) Returns the smallest (largest) value among the given numeric values.
(sqrt n) Square root.
(expt n1 n2) Computes n^n2
(floor n) Returns the largest integer not greater than n.
(ceiling n) Returns the smallest integer not less than n.

Higher-Order Procedures

(map f lst 1 lst2 . . .)
Apply f to each element of all the input lists, in some order, and collect the return values into a list, which is returned from map.

Chars and Strings

(char? obj) It returns #t if obj is a character.
(char=? c1 c2) It returns #t if c1 and c2 are the same character.
(char->integer c) It converts c to the corresponding integer (character code).
Example: (char->integer #\a) -> 97
(integer->char n) It converts an integer to the corresponding character.
(char<? c1 c2)
(char<=? c1, c2)
(char> c1 c2)
(char>= c1 c2)
These functions compare characters. Actually, the functions compare the size of the character codes. For instance, (char<? c1 c2) is equal to (< (char->integer c1) (char->integer c2)) .

(string? s) It returns #t if s is a string.
(make-string n c) It returns a string consisting of n of characters c. The character c can be omitted.
(string-length s) It returns the length of a string s.
(string=? s1 s2) It returns #t if strings s1 and s2 are the same.
(string-ref s idx) It returns the idx-th character (counting from 0) of a string s.
(substring s start end) It returns a substring of s consisting of characters from start to (end-1).
(substring "abcdefg" 1 4) -> "bcd"
(string-append s1 s2 ...) It connects strings s1, s2 ....

Misc

```
(list (char->integer #\A) (char->integer #\Z) (char->integer #\a) (char->integer #\z))
'(65 90 97 122)
```