Correttezza: induzione ed invarianti di ciclo

## Esercizio 1

Si consideri il seguente algoritmo che calcola il quadrato del suo parametro:

Square(z)

1: ▷ Pre: z numero intero ≥ 0

2: ▷ Post: ritorna l’intero z2

3: x ← 0

4: y ← 0

5: while x < z do

6: y ← y + 2x + 1

7: x ← x + 1

8: return y

Si dimostri formalmente la correttezza della funzione, cioè:

* si trovi l’invariante del ciclo,
* si dimostri che l’invariante è vero inizialmente e viene mantenuto dal ciclo
* usando l’invariante si dimostri che il valore restituito è z2.

1. Invariante = condizione mantenuta all’interno del ciclo + condizione di uscita dal ciclo;

In questo caso:

1. All’inizio quindi

Per ogni iterazione quindi

, che soddisfa

1. Prendendo per esempio , il ciclo termina con , e avremo

## Esercizio 2

Nota: l’invariante non è altro che il caso base

k = n

z = 1

y = x

while k > 0

if x % 2 == 1

z = z \* y

y = y^2

k = k/2

return z

FastExp(x, n)

if(n == 0)

return 1

else

y = FastExp(x, n/2)

if(n % 2 == 1)

return y^2 \* x

else

return y^2

## Esercizio 3

Ndr: pre condizione quello che deve entrare dentro il ciclo

post condizione quello che esce

LinSearch(A[i…j], a)

//pre-con: A array int, a intero

//post-con: k = A[k] se esiste, -1 altrimenti

//invariante: per ogni i < k-1, A[i] != a

k = 0

while k < max

if A[k] == a

return A[k]

else

k++

return -1

LinSearchRec(A[i…j], k)

//pre-con: A array int, a intero

//post-con: k = A[k] se esiste, -1 altrimenti

if(i < j)

if(i == k && A[i] != nil)

return A[i]

else

return LinSearchRec(A[], i++, j, k)

else

return -1

## Esercizio 4

DicSearchRec(A[i…j], a)

//Pre: A array interi, a intero

//Post: k A[i…j] = a se esiste, -1 altrimenti

if(i==j)

if(A[i]==a)

return i

else

return -1

else

if(A[j/2]<a)

return DicSearchRec(A[i…j/2], a)

else

if(A[j/2]==a)

return j/2

else

return DicSearchRec(A[j/2…j], a)

DicSearchIter(A[i…j], a)

//Pre: A array interi, a intero

//Post: k A[i…j] == a se esiste, -1 altrimenti

//invariante: se A[h]==a, allora h i…j

y = i

z = j

while y!=z

if(A[y]==a)

return y

else

if(A[z]==a)

return z

else

if(a < A[z])

z = (y+z)/2

else

y = (y+z)/2

if(A[y]==a)

return y

else

return -1

## Esercizio 5

1. inv: A[j] = min(A[j…n])

All’inizio A[j] == A[n] quindi banalmente valido; per ogni iterazione, A[j] viene spostato indietro se minore del precedente, quindi sarà il minimo tra esso e i suoi seguenti.

1. inv: A[i…n] >= A[1…i-1]

Valido sempre perché è ordinato in senso non decrescente, e valido di base perché A[1…0] = quando i = 1.

Alberi

Negli esercizi che seguono si assume che gli alberi binari siano realizzati con record di tre campi: key per la chiave (di solito un intero), left e right per i puntatori al sottoalbero sinistro e destro rispettivamente. Se invece gli alberi sono k-ari allora la realizzazione utilizza record a tre campi con il campo key come sopra, il campo child per il puntatore al primo dei sottoalberi, il campo sibling per il puntatore al fratello.

## Esercizio 1

Foo(T)

parentK <- T.key

success <- true

C <- T.child

**while** C != nil **do**

success <- SumSiblings(C, parentK) Tree-DFS(C)

C <- C.sibling

**end while**

**return** success

SumSiblings(C, p)

sum <- C.key

C <- C.sibling

**while** C != nil **do**

sum <- sum + C.key

C <- C.sibling

**end while**

**if** sum != p **then**

**return** false

**else**

**return** true

**end if**

## Esercizio 2

SommaRamo(T, k)

**if** k == 0 **then**

k <- SommaCammino(T)

**end if**

C <- T.child

**while** C != nil do

SommaRamo(C, k)

**if** C.child == nil **then**

C.child <- nuova foglia V

V.key <- k

V.child <- V.sibling <- nil

**else**

C <- C.sibling

**end if**

**end while**

SommaCammino(T)

k <- T.key

C <- T.child

**while** C != nil **do**

k <- k + C.key + SommaCammino(C)

C <- C.sibling

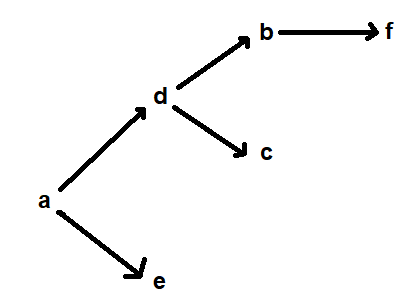
**end while**

**return** k

Grafi

## Esercizio 1

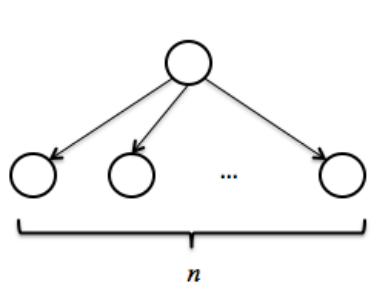
1.



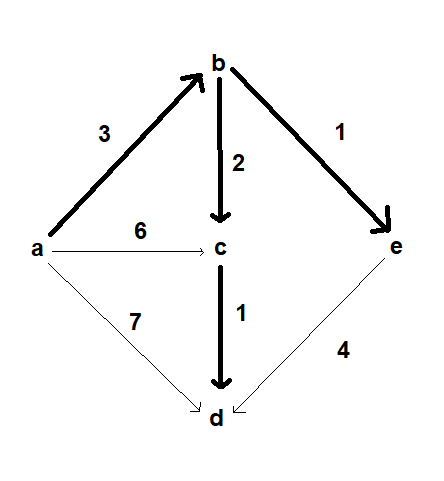
1. a (radice)
2. de (visita a, in coda d ed e)
3. ebc (visita b, in coda b e c, e già in coda)
4. bc (visita e, niente da aggiungere alla coda, a e d già visitate)
5. cf (visita b, in coda f, a ed e già visitate, c già in coda)
6. f (visita c, niente da aggiungere alla coda, a ed e già visitate)
7. ∅ (visita f, niente da aggiungere alla coda, d ed e già visitate)

2.

è un albero la cui radice r ha n figli:



## Esercizio 2



NB: Dijkstra lavora su uno heap minimo (per ogni elemento n < dei figli, figlio sinistro 2n > figlio destro 2n+1)

1. a0, b∞, c∞, d∞, e∞
2. b3, d7, c6, e∞ (estrazione a; aggiornamento b, c, d; riordinamento: d>c quindi d è figlio sx di b)
3. e1, d7, c5 (estrazione b; aggiornamento c, e; riordinamento: e nuovo minimo)
4. c5, d7 (estrazione e; riordinamento: c nuovo minimo)
5. d6 (estrazione c; aggiornamento d)

Esercizi da fonti esterne

## N.1

Si consideri lo heap minimo rappresentato con l’array:

[14, 32, 18, 50, 41, 23, 90, 87, 64, 53, 43]

ottenuto dopo l’estrazione del minimo **5**, dall’array:

1. [5, 14, 18, 32, 41, 23, 90, 50, 64, 53, 43, 87]
2. [5, 14, 23, 32, 41, 18, 90, 50, 64, 53, 43, 87]
3. [5, 14, 18, 50, 41, 23, 90, 32, 64, 53, 43, 87]

Da questo, dopo l’inserimento della chiave **15** si ottiene:

1. [14, 32, 15, 50, 41, 18, 90, 87, 64, 53, 43, 23]
2. [14, 32, 15, 50, 41, 23, 90, 87, 64, 53, 43, 18]
3. [14, 50, 15, 32, 41, 18, 90, 87, 64, 53, 43, 23]

### Soluzione:

In un heap minimo, l’etichetta dei figli di ogni nodo (radice compresa) sono > dell’etichetta del padre.

I figli di ogni nodo sono posizionati a 2i (figlio sinistro) e 2i+1 (figlio destro).

Allora, si può notare che le configurazioni **1** e **1** sono le uniche che lasciano lo heap in uno stato corretto.

## N.2

Si consideri il seguente algoritmo:

Casper(A[0…n-1])

> Pre: array di interi

for i := 0 to n-2

for j := i+1 to n-1

if A[i] = A[j] then

return false

return true

Si risponda succintamente alle seguenti domande:

1. Quando Casper(A) ritorna true?
2. Qual è la sua complessità in termini di 𝚯?
3. Sapreste indicare un algoritmo che risolva lo stesso problema in tempo asintoticamente migliore?

### Soluzione:

1. Ritorna true se tutti i numeri interi nell’array A sono diversi;
2. 𝚯(n2);
3. Serve ordinare l’array con un algoritmo come MergeSort o HeapSort per portare l’algoritmo ad una complessità di 𝚯(n log n).