ASD Cheatsheet

alessandro.manfucci@studenti.unitn.it

10/01/2023

Table of contents

1 Pseudo Linguaggio

- a = b
- a $\leftrightarrow$ b $\equiv$ tmp = a; a = b; b = tmp
- T [ ] A = new T [1…n]
- T [ ][ ] B = new T [1…n][1…m]
- int, float, boolean
- and, or, not
- ==, $\neq$, $\leq$, $\geq$
- +, -, $\cdot$, /, $\lfloor x \rfloor$, $\lceil x \rceil$, log, x^2, mod, …
- iif(condizione, v_1, v_2)
- if condizione then istruzione
- if condizione then istruzione1 else istruzione2
- while condizione do istruzione
- foreach elemento $\in$ insieme do istruzione
- return
- % commento
- for indice = estremoInf to estremoSup do istruzione

- int indice = estremoInf
  while indice $\leq$ estremoSup do
  | istruzione
  | indice = indice + 1
  -

- for indice = estremoSup downto estremoInf do istruzione

- int indice = estremoSup
  while indice $\geq$ estremoInf do
  | istruzione
  | indice = indice - 1
  -

RETTANGOLO
int lunghezza
int altezza

- rettangolo r = new rettangolo
- r.altezza = 10
- delete r
- r = nil

2 Strutture dati elementari

2.1 Pila

STACK

```
STACK STACK()
% Restituisce true se la pila è vuota
boolean isEmpty()
% Inserisce v in cima alla pila
push(ITEM v)
% Rimuove l'elemento in cima alla pila e lo restituisce
ITEM pop()
% Legge l'elemento in cima alla pila
ITEM top()
```

2.2 Coda

QUEUE

```
QUEUE QUEUE()
% Restituisce true se la coda è vuota
boolean isEmpty()
% Inserisce v in fondo alla coda
enqueue(ITEM v)
% Estrae l'elemento in testa alla coda e lo restituisce al chiamante
ITEM dequeue()
% Legge l'elemento in testa alla coda
ITEM top()
```

3 Strutture dati astratte

3.1 Sequenza

SEQUENCE

```
SEQUENCE SEQUENCE()
% Restituisce true se la sequenza è vuota
boolean isEmpty()
% Restituisce true se p = pos0 o se p = posn+1
boolean finished(POS p)
% Restituisce la posizione del primo elemento
POS head()
% Restituisce la posizione dell'ultimo elemento
POS tail()
% Restituisce la posizione dell'elemento che segue p
POS next(POS p)
% Restituisce la posizione dell'elemento che precede p
POS prev(POS p)
% Inserisce l'elemento v di tipo ITEM nella posizione p e restituisce
% la posizione del nuovo elemento, che diviene il predecessore di p
POS insert(POS p, ITEM v)
% Rimuove l'elemento contenuto nella posizione p e restituisce la posizione
% del successore di p, che diviene il successore del predecessore di p
POS remove(POS p)
% Legge l'elemento di tipo ITEM contenuto nella posizione p
ITEM read(POS p)
% Scrive l'elemento v di tipo ITEM nella posizione p
write(POS p, ITEM v)
```

3.2 Insieme

SET

```
SET SET()
% Restituisce la cardinalità dell'insieme
int size()
% Restituisce true se x è contenuto nell'insieme
boolean contains(ITEM x)
% Inserisce x nell'insieme, se non è giù presente
insert(ITEM x)
% Rimuove x dall'insieme, se è presente
remove(ITEM x)
% Restituisce un nuovo insieme che è l'unione di A e B
Set union(Set A, Set B)
% Restituisce un nuovo insieme che è l'intersezione di A e B
Set intersection(Set A, Set B)
% Restituisce un nuovo insieme che è la differenza di A e B
Set difference(Set A, Set B)
```

Sia n il numero di elementi nell'insieme e m la capacità dell'insieme.

Implementazione	contains()	insert()	remove()	min()	foreach()
Array booleano	O(1)	O(1)	O(1)	O(m)	O(m)
Lista non ordinata	O(n)	O(n)	O(n)	O(n)	O(n)
Lista ordinata	O(n)	O(n)	O(n)	O(1)	O(n)
Array ordinato	O(log n)	O(n)	O(n)	O(1)	O(n)
RB Tree	O(log n)	O(log n)	O(log n)	O(log n)	O(n)
Hash Table	O(1)	O(1)	O(1)	O(m)	O(m)

3.3 Dizionario

`DICTIONARY`

```
DICTIONARY DICTIONARY()
% Restituisce il valore associato alla chiave k se presente, nil altrimenti
ITEM lookup(ITEM k)
% Associa il valore v alla chiave k
insert(ITEM k, ITEM v)
% Rimuove l'associazione della chiave k
remove(ITEM k)
```

Implementazione	lookup()	insert()	remove()	foreach()
Array non ordinato	O(n)	O(1),O(n)	O(1)	O(n)
Array ordinato	O(log n)	O(n)	O(n)	O(n)
Lista non ordinata	O(n)	O(1),O(n)	O(n)	O(n)
RB Tree	O(log n)	O(log n)	O(log n)	O(n)
Hash Table	O(1)	O(1)	O(1)	O(n)

3.4 Priority Queue

`MIN-PRIORITYQUEUE`

```
MIN-PRIORITYQUEUE MIN-PRIORITYQUEUE()
% Crea una coda a priorità con capacità n
PRIORITYQUEUE PriorityQueue(int n)
% Restituisce true se la coda a priorità è vuota
boolean isEmpty()
```

```
% Restituisce l'elemento minimo di una coda a priorità non vuota
ITEM min()
% Rimuove e restituisce l'elemento minimo di una coda a priorità non vuota
deleteMin()
% Inserisce l'elemento x con priorità p nella coda a priorità e restituisce
% un oggetto PRIORITYITEM che identifica x all'interno della coda
PRIORITYITEM insert(ITEM x, int p)
% Diminuisce la priorità dell'oggetto identificato da y portandola a p
decrease(PRIORITYITEM y, int p)
```

Implementazione	min()	deleteMin()	insert()	decrease()
Array/Lista non ordinato	O(n)	O(n)	O(n)	O(n)
Array ordinato	O(1)	O(n)	O(n)	O(log n)
Lista ordinata	O(1)	O(1)	O(n)	O(n)
RB Tree	O(log n)	O(log n)	O(log n)	O(log n)
Heap Tree	O(1)	O(log n)	O(log n)	O(log n)

3.5 Albero binario

TREE

```
TREE TREE()
% Costituisce un nuovo nodo, contenente v, senza figli o genitori
TREE(ITEM v)
% Legge il valore memorizzato nel nodo
ITEM read()
% Modifica il valore memorizzato nel nodo
write(ITEM v)
% Restituisce il padre, oppure nil se questo è il nodo radice
TREE parent()
% Restituisce il figlio sinistro di questo nodo, oppure nil se è assente
TREE left()
% Restituisce il figlio destro di questo nodo, oppure nil se è assente
TREE right()
% Inserisce il sottoalbero radicato t come figlio sinistro di questo nodo
insertLeft(TREE t)
% Inserisce il sottoalbero radicato t come figlio destro di questo nodo
insertRight(TREE t)
% Distrugge ricorsivamente il figlio sinistro di questo nodo (in O(n) con punt.)
deleteLeft()
% Distrugge ricorsivamente il figlio destro di questo nodo (in O(n) con punt.)
deleteRight()
```

3.6 Albero generico

TREE

```
TREE TREE()
% Costituisce un nuovo nodo, contenente v, senza figli o genitori
TREE(ITEM v)
% Legge il valore memorizzato nel nodo
ITEM read()
% Modifica il valore memorizzato nel nodo
write(ITEM v)
% Restituisce il padre, oppure nil se questo è il nodo radice
TREE parent()
% Restituisce il primo figlio da sinistra, oppure nil se questo nodo
% è una foglia
TREE leftmostChild()
% Restituisce il primo fratello sulla destra, oppure nil se è assente
TREE rightSibling()
% Inserisce il sottoalbero t come primo figlio di questo nodo
insertChild(TREE t)
% Inserisce il sottoalbero t come prossimo fratello di questo nodo
insertSibling(TREE t)
% Distrugge l'albero radicato identificato dal primo figlio
deleteChild()
% Distrugge l'albero radicato identificato dal prossimo fratello
deleteSibling()
% Distrugge l'albero radicato identificato dal nodo
delete(TREE t)
```

3.7 Grafo

`GRAPH`

```
GRAPH GRAPH()
% Restituisce l'insieme di tutti i vertici
SET V()
% Restituisce il numero di nodi
int size()
% Restituisce l'insieme dei nodi adiacenti ad u
SET adj(NODE u)
% Aggiunge un nodo u al grafo
insertNode(NODE u)
% Aggiunge l'arco (u, v) al grafo
insertEdge(NODE u, NODE v)
% Rimuove il nodo u dal grafo (in O(n) con vettori/liste adiacenza)
removeNode(NODE u)
% Rimuove l'arco (u, v) dal grafo
removeEdge(NODE u, NODE v)
```

4 Algoritmi ordinamento

4.1 selectionSort

- handout.2-analisi.49/65

Pessimo	Medio	Ottimo
$O(n^2)$	$O(n^2)$	$O(n^2)$

`selectionSort(ITEM[ ] A, int n)`

```
for i = 1 to n - 1 do
 | int min = min(A, i, n)
 | A[i] <-> A[min]
 -
```

4.1.1 min

- O(n)

```
int min(ITEM[ ] A, int i, int n)
```

```
% Posizione del minimo parziale
int min = i
for j = i + 1 to n do
 | if A[j] < A[min] then
 |  | % Nuovo minimo parziale
 |  |  min = j
 |  -
 -
return min
```

4.2 insertionSort

Pessimo	Medio	Ottimo
O(n^2)	O(n^2)	O(n)

- handout.2-analisi.52/65

```
insertionSort(ITEM[] A, int n)
```

```
for i = 2 to n do
 | ITEM temp = A[i]
 | int j = i
 | while j > 1 and A[j - 1] > temp do
 |  | A[j] = A[j - 1]
 |  | j = j - 1
 |  -
 -
 A[j] = temp
```

4.3 mergeSort

Pessimo	Medio	Ottimo
O(n · log n)	O(n · log n)	O(n · log n)

- handout.2-analisi.61/65

```
mergeSort(ITEM A[ ], int first, int last)
```

```
if first < last then
 | int mid = b(first + last)/2c
 | mergeSort(A, first, mid)
 | mergeSort(A, mid + 1, last)
 | merge(A, first, last, mid)
 -
```

4.3.1 merge

- O(n)

```
merge(ITEM A[ ], int first, int last, int mid)
```

```
int i, j, k, h
i = first
j = mid + 1
k = first
while i <= mid and j <= last do
 | if A[i] <= A[j] then
 |  | B[k] = A[i]
 |  | i = i + 1
 | else
 |  | B[k] = A[j]
 |  | j = j + 1
 |  -
 |  k = k + 1
 -
j = last
for h = mid downto i do
 | A[j] = A[h]
 | j = j - 1
 -
for j = first to k - 1 do
 | A[j] = B[j]
 -
```

4.4 quickSort

- handout.12-divide.12/34

Pessimo	Medio	Ottimo
O(n^2)	O(n · log n)	O(n · log n)

```
quickSort(ITEM[] A, int n)
```

4.5 heapSort

Pessimo	Medio	Ottimo
O(n · log n)	O(n · log n)	O(n · log n)

- handout.10-strutture-speciali.16/64

```
heapSort(ITEM[] A, int n)
```

```
heapBuild(A, n)
for i=n downto 2 do
 | swap(A, 1, i) % L'elemento massimo viene spostato in fondo
 | maxHeapRestore(A, 1, i-1)
 -
```

4.5.1 maxHeapRestore

- O(log n)

```
maxHeapRestore(ITEM[] A, int i, int dim)
```

```
int max = i % Sceglie la radice
if l(i) <= and dim A[l(i)] > A[max] then
 | max = l(i)
if r(i) <= and dim A[r(i)] > A[max] then
 | max = r(i)
if i != max then % Se i == max l'albero è apposto
 | swap(A, i, max) % Scambia la radice e il maggiore tra i suoi figli
 | maxHeapRestore(A, max, dim) % Controlla il sottoalbero con radice max
 -
```

4.5.2 heapBuild

- O(n)

```
heapBuild(ITEM[] A, int n)
```

```
for i = floor(n/2) downto 1 do
 | maxHeapRestore(A, i, n)
 -
```

5 Algoritmi alberi binari

5.1 dfs

- O(n)

```
dfs(TREE t)
```

```
if t != nil then
 | % pre-order visit of t
 | print t
 | dfs(t.left())
 | % in-order visit of t
 | print t
 | dfs(t.right())
```

```
  | % post-order visit of t
  | print t
  -
```

5.2 lookupNode

Pessimo	Medio	Ottimo
O(n)	O(log n)	O(log n)

```
TREE lookupNode(TREE T, ITEM k)
```

```
if T == nil or T.key == k then
 | return T
else
 | return lookupNode(iif(k < T.key, T.left, T.right), k)
 -
```

5.3 min

Pessimo	Medio	Ottimo
O(n)	O(log n)	O(log n)

```
TREE min(TREE T)
```

```
TREE u = T
while u.left != nil do
 | u = u.left
 -
return u
```

5.4 max

Pessimo	Medio	Ottimo
O(n)	O(log n)	O(log n)

```
TREE max(TREE T)
```

```
TREE u = T
while u.right != nil do
 | u = u.right
 -
return u
```

5.5 predecessorNode

Pessimo	Medio	Ottimo
O(n)	O(log n)	O(log n)

```
TREE predecessorNode(TREE t)
```

```
if t == nil then
 | return t
 -
if t.left != nil then % Caso 1
 | return max(t.left)
else % Caso 2
 | TREE p = t.parent
 | while p != nil and t == p.left do
 |  | t = p
 |  | p = p.parent
 |  -
 | return p
 -
```

5.6 successorNode

Pessimo	Medio	Ottimo
O(n)	O(log n)	O(log n)

```
TREE successorNode(TREE t)
```

```
if t == nil then
 | return t
 -
if t.right != nil then % Caso 1
 | return min(t.right)
else % Caso 2
 | TREE p = t.parent
 | while p != nil and t == p.right do
 |  | t = p
 |  | p = p.parent
 |  -
 | return p
 -
```

5.7 insertNode

Pessimo	Medio	Ottimo
O(n)	O(log n)	O(log n)

```
TREE insertNode(TREE T, ITEM k, ITEM v)
```

```
TREE p = nil % Padre
TREE u = T
while u != nil and u.key != k do % Cerca posizione inserimento
 | p = u
 | u = iif(k < u.key, u.left, u.right)
 -
if u != nil and u.key == k then
 | u.value = v % Chiave già presente
else
 | TREE new = TREE(k, v) % Crea un nodo coppia chiave-valore
 | link(p, new, k)
 | if p == nil then
 |  | T = new % Primo nodo ad essere inserito
 |  -
 -
return T % Restituisce albero non modificato o nuovo nodo
```

```
link(TREE p, TREE u, ITEM k)
```

```
if u != nil then
 | u.parent = p % Registrazione padre
 -
if p != nil then
 | if k < p.key then
 |  | p.left = u % Attaccato come figlio sinistro
 | else
 |  | p.right = u % Attaccato come figlio destro
 |  -
 -
```

5.8 removeNode

Pessimo	Medio	Ottimo
O(n)	O(log n)	O(log n)

```
TREE removeNode(TREE T, ITEM k)
```

5.9 isRedBlack

- O(n)
- soluzione.19-08-22.A2

5.9.1 Proprietà RB Tree

1. La radice è nera
2. Tutte le foglie sono nere
3. Entrambi i figli di un nodo rosso sono neri
4. Ogni cammino semplice da un nodo ad una delle sue foglie ha sempre lo stesso numero di nodi neri (ovvero ogni nodo nel suo sottoalbero ha i figli con la stessa altezza nera)

```
boolean isRedBlack(TREE T)
```

```
% Proprietà (1)
if T.color == red then
 | return false
else
 | return (blackHeight(T) > 0)
 -
```

```
int blackHeight(TREE T)
```

```
% Proprietà (2)
if T == nil then
 | return iif(T.color == red, -1, 1)
 -
% Proprietà (3)
if t.color == red and t.parent != nil and t.parent.color == red then
 | return -1
 -
% Proprietà (4)
int bhL = blackHeight(T.left)
int bhR = blackHeight(T.right)
if bhL < 0 or bhR < 0 or bhL != bhR then
 | return -1
else
 | return bhL + iif(t.color == black, 1, 0)
 -
```

6 Algoritmi alberi generici

6.1 dfs

- O(n)

```
dfs(TREE t)
```

```
if t != nil then
 | % pre-order visit of node t
 | print t
 | TREE u = t.leftmostChild()
 | while u != nil do
 |  | dfs(u)
 |  | u = u.rightSibling()
 |  -
 | % post-order visit of node t
 | print t
 -
```

6.2 bfs

- O(n)

```
bfs(TREE t)
```

```
QUEUE Q = QUEUE()
Q.enqueue(t)
while not Q.isEmpty() do
 | TREE u = Q.dequeue()
 | % visita per livelli nodo u
 | print u
 | u = u.leftmostChild()
 | while u != nil do
 |  | Q.enqueue(u)
 |  |u = u.rightSibling()
 |  -
 -
```

7 Algoritmi grafi non orientati

7.1 cc

- O(n+m)
- handout.09-grafi.53/101

```
int[] cc(GRAPH G)
```

```
int[] id = new int[1...G.size()]
foreach u in G.V() do
 | id[u] = 0
 -
int counter = 0
foreach u in G.V() do
 | if id[u] == 0 then
 |  | counter = counter + 1
 |  | ccdfs(G, counter , u, id)
 |  -
 -
 return id
```

```
ccdfs(GRAPH G, int counter , NODE u, int[] id)
```

```
id[u] = counter
foreach v in G.adj(u) do
if id[v] == 0 then
 | ccdfs(G, counter , v, id)
 -
```

7.2 hasCycle

- O(n+m)
- handout.09-grafi.58/101

```
boolean hasCycle(GRAPH G)
```

```
boolean[] visited = new boolean[1...G.size()]
foreach u in G.V() do
 | visited[u] = false
 -
foreach u in G.V() do
 | if not visited[u] then
 |  | if hasCyclerec(G, u, null, visited) then
 |  |  | return true
 |  |  -
 |  -
 -
return false
```

```
boolean hasCycleRec(GRAPH G, NODE u, NODE p, boolean[] visited)
```

```
visited[u] = true
foreach v in G.adj(u)\{p} do
 | if visited[v] then
 |  | return true
 | else if hasCycleRec(G, v, u, visited) then
 |  | return true
 |  -
 -
return false
```

8 Algoritmi grafi orientati

8.1 bfs

- O(n+m)
- handout.09-grafi.41/101

```
bfs(GRAPH G, NODE r, NODE[] parent)
```

```
% albero bfs su vettore padri
QUEUE Q = Queue( )
S.enqueue(r)
boolean[] visited = new boolean[1...G.size()]
foreach u in G.V()\{r} do
 | visited[u] = false
 -
visited[r] = true
parent[r] = nil
while not Q.isEmpty() do
 | NODE u = Q.dequeue()
 | % visita il nodo u
 | foreach v in G.adj(u) do
 |  | % visita l'arco (u, v)
 |  | if not visited[v] then
```

```
 |  |  | visited[v] = true
 |  |  | parent[v] = u
 |  |  | Q.enqueue(v)
 |  |  -
 |  -
 -
```

8.2 distance

- O(n+m)
- handout.09-grafi.39/101

`distance(GRAPH G, NODE r, int[] distance)`

```
QUEUE Q = QUEUE()
Q.enqueue(r)
foreach u in G.V()\{r} do
 | distance[u] = inf
 -
distance[r] = 0
while not Q.isEmpty() do
 | NODE u = Q.dequeue()
 | foreach v in G.adj(u) do
 |  | if distance[v] == inf then % Se il nodo v non è stato scoperto
 |  |  | distance[v] = distance[u] + 1
 |  |  | Q.enqueue(v)
 |  |  -
 |  -
 -
```

8.3 dfs

- O(n+m)
- handout.09-grafi.45/101

`dfs(GRAPH G, NODE u, boolean[] visited)`

```
visited[u] = true
% visita il nodo u (pre-order)
foreach v in G.adj(u) do
 | if not visited[v] then
 |  | % visita l'arco (u, v)
 |  | dfs(G, v, visited)
 |  -
 -
% visita il nodo u (post-order)
```

8.4 dfs-schema

- O(n+m)
- handout.09-grafi.63/101

```
dfs-schema(GRAPH G, NODE u, int &time, int[ ] dt, int[ ] ft)
```

```
% visita il nodo u (pre-order)
time = time + 1; dt[u] = time
foreach v in G.adj(u) do
 | % visita l'arco (u, v) (qualsiasi)
 | if dt[v] == 0 then
 |  | % visita l'arco (u, v) (albero)
 |  | dfs-schema(G, v, time, dt, ft)
 | else if dt[u] > dt[v] and ft[v] == 0 then
 |  | % visita l'arco (u, v) (indietro)
 | else if dt[u] < dt[v] and ft[v] 6= 0 then
 |  | % visita l'arco (u, v) (avanti)
 | else
 |  | % visita l'arco (u, v) (attraversamento)
 |  -
 -
% visita il nodo u (post-order)
time = time + 1; ft[u] = time
```

8.5 hasCycle

- O(n+m)
- handout.09-grafi.71/101

```
boolean hasCycle(GRAPH G)
```

```
boolean[] visited = new boolean[1...G.size()]
foreach u in G.V() do
 | visited[u] = false
 -
foreach u in G.V() do
 | if not visited[u] then
 |  | if hasCyclerec(G, u, null, visited) then
 |  |  | return true
 |  |  -
 |  -
 -
return false
```

```
boolean hasCycleRec(GRAPH G, NODE u, int &time, int[] dt, int[] ft)
```

```
time = time + 1; dt[u] = time
foreach v in G.adj(u) do
 | if dt[v] == 0 then
 |  | if hasCycleRec(G, v, time, dt, ft) then
 |  |  | return true
 | else if dt[u] > dt[v] and ft[v] == 0 then
 |  | return true
 |  -
 -
```

```
time = time + 1; ft[u] = time
return false
```

8.6 topSort

- O(n+m)
- handout.09-grafi.76/101

```
STACK topSort(GRAPH G)
```

```
STACK S = STACK()
boolean[] visited = boolean[1...G.size()]
foreach u in G.V() do visited[u] = false
foreach u in G.V() do
 | if not visited[u] then
 |  | ts-dfs(G, u, visited, S)
 |  -
 -
return S
```

```
ts-dfs(GRAPH G, NODE u, boolean[] visited, STACK S)
```

```
visited[u] = true
foreach v in G.adj(u) do
 | if not visited[v] then
 |  | ts-dfs(G, v, visited, S)
 |  -
 -
S.push(u)
```

8.7 transpose

- O(n+m)
- handout.09-grafi.86/101

```
GRAPH transpose(GRAPH G)
```

```
GRAPH GT = GRAPH()
foreach u in G.V() do
 | GT.insertNode(u)
 -
foreach u in G.V() do
 | foreach v in G.adj(u) do
 |  | GT.insertEdge(v, u)
 |  -
 -
return GT
```

8.8 scc

- O(n+m)

- handout.09-grafi.83/101

`int[] scc(GRAPH G)`

```
STACK S = topSort(G) % First visit O(n+m)
GT = transpose(G) % GRAPH transposal O(n+m)
return cc(GT, S) % Second visit O(n+m)
```

`cc(GRAPH G, STACK S)`

```
int[] id = new int[G.size()]
foreach u in G.V() do
 | id[u] = 0
 -
int counter = 0
while not S.isEmpty() do
 | u = S.pop()
 | if id[u] == 0 then
 |  | counter = counter + 1
 |  | ccdfs(G, counter , u, id)
 |  -
 -
return id
```

`ccdfs(GRAPH G, int counter , NODE u, int[ ] id)`

```
id[u] = counter
foreach v in G.adj(u) do
 | if id[v] == 0 then
 |  | ccdfs(G, counter , v, id)
 |  -
 -
```

9 Algoritmi divide-et-impera

9.1 binarySearch

- O (log n)
- handout.01-introduzione.20/27

`int binarySearch(int[] A, int v, int i, int j)`

```
if i > j then
 | return 0
else
 | int m = floor((i+j)/2)
 | if S[m] == v then
 |  | return m
 | else if S[m] < v then
```

```
 |  | return binarySearch(S, v, m+1, j)
 | else
 |  | return binarySearch(S, v, i, m-1)
 |  -
 -
```

- $\forall$ $binarySearch(A, v, i, j), v \in A \Leftrightarrow v \in A[i...j]$

9.2 searchFirst

- O(log n)

```
int searchFirst(int[] A, int v, int i, int j)
```

```
if i > j then
 | return 0
else if i == j and A[i] == v then
 | return i
else
 | int m = floor((i+j)/2)
 | if A[m] < v then
 |  | return searchFirst(A, v, m+1, j)
 | else
 |  | return searchFirst(A, v, i, m)
 |  -
 -
```

- $\forall$ $searchFirst(A, v, i, j), \forall h \in A[1...i-1], h < v \ \wedge \ \forall k \in A[i...j], k \geq v$

Con questo algoritmo troviamo la **prima** (più a sinistra) occorrenza di v. Procediamo quindi per induzione.

Base: *con $i = 1, j = n$ vale $A[i...j] = A$, dunque $A[1...0] = \emptyset = A[n...n+1]$ ed entrambe le condizioni sono soddisfatte*

Passo induttivo: *Supponiamo che per $binarySearch(A, v, i, j)$ valga l'invariante e dimostriamo che vale anche per l'invocazione successiva, $binarySearch(A, v, i', j')$*

1) Se $i > j$ allora $A[i...j] = \emptyset$ e la funzione termina
2) Se $i = j$ allora $v \in A \Leftrightarrow v = A[i]$
Posto $m := \lfloor (i+j)/2 \rfloor$
3) Se $A[m] < v$ allora $i' := m+1, j' := j$ e certamente il passo induttivo è verificato
4) Se $A[m] \geq v$ allora $i' := i, j' := m$ e certamente il passo induttivo è verificato

9.3 searchLast

- O(log n)

```
int searchLast(int[] A, int v, int i, int j)
```

```
if i > j then
 | return 0
else if i == j and A[i] == v then
 | return i
else
 | int m = ceil((i+j)/2)
```

```
 | if A[m] <= v then
 |  | return searchLast(A, v, m, j)
 | else
 |  | return searchLast(A, v, i, m-1)
 |  -
 -
```

- $\forall\ searchLast(A, v, i, j), \forall h \in A[j+1...n], h > v\ \wedge\ \forall k \in A[i...j], k \geq v$

Con questo algoritmo troviamo l'**ultima** (più a destra) occorrenza di v.

9.4 hanoi

- O(2^n)
- handout.12-divide.10/34

`hanoi(int n, int src, int dest, int middle)`

```
if n == 1 then
 | print src -> dest
else
 | hanoi(n - 1, src, middle, dest)
 | print src -> dest
 | hanoi(n - 1, middle, dest, src)
 -
```

10 Algoritmi misc.

10.1 maxSum

- O(n)

`maxSum(int[] A, int n)`

```
maxSoFar = 0
maxHere = 0
for i=1 to n do
 | maxHere = max(0, maxSoFar+A[i])
 | maxSoFar = max(maxSoFar, maxHere)
return maxSoFar
```

11 Studio equazioni di ricorrenza

11.0.1 Template

$T(n) = a_1 T(n/b_1) + a_2 T(n/b_2) + f(n)$

i) $T(n)$ *crescente e positiva*
ii) $T(n) \geq f(n) \implies T(n) = \Omega(f(n))$
iii) $T(n) \leq (a_1 + a_2)T'(n/min(b_1, b_2)) \implies T(n) = O(...)$ *per il MT*
iv) $T(n) \geq (a_1 + a_2)T'(n/max(b_1, b_2)) \implies T(n) = \Omega(...)$ *per il MT*
v) Vogliamo dimostrare che $T(n) = O(f(n))$*, ovvero che con* $c > 0, m \geq 0$ *vale* $T(n) \leq cf(n) \forall n \geq m$

- *Base:*

- *Ipotesi induttiva:*
- *Passo induttivo:*

11.1 Metodo dell'albero di ricorsione

Livello	Dim. input	Costo per chiamata	N. chiamate	Costo livello

11.2 Master Theorems

TEO.1 Ricorrenze lineari con partizione bilanciata

Siano a e b costanti intere tali che $a \geq 1$ e $b \geq 2$. Siano poi c e β costanti reali tali che $c > 0$ e $\beta \geq 0$. Sia $T(n)$ una funzione di ricorrenza della seguente forma:

$$T(n) = \begin{cases} aT(n/b) + cn^{\beta} & \text{se} \quad n > 1 \\ d & \text{se} \quad n \leq 1 \end{cases} \tag{1}$$

Allora, posto $\alpha := \frac{\log a}{\log b} = \log_b a$ vale:

$$T(n) = \begin{cases} \Theta(n^{\alpha}) & \text{se} \quad \alpha > \beta \\ \Theta(n^{\alpha} \log n) & \text{se} \quad \alpha = \beta \\ \Theta(n^{\beta}) & \text{se} \quad \alpha < \beta \end{cases} \tag{2}$$

TEO.2 Ricorrenze lineari con partizione bilanciata – Est.

Siano $a \geq 1$, $b > 1$ e $f(n)$ una funzione asintoticamente positiva. Sia poi $T(n)$ una funzione di ricorrenza della seguente forma:

$$T(n) = \begin{cases} aT(n/b) + f(n) & \text{se} \quad n > 1 \\ d & \text{se} \quad n \leq 1 \end{cases} \tag{3}$$

Allora vale:

1) *Se $\exists \epsilon > 0 : f(n) = O(n^{\log_b(a) - \epsilon})$*
   allora $T(n) = \Theta(n^{\log_b a})$
2) *Se $f(n) = \Theta(n \log_b a)$*
   allora $T(n) = \Theta(f(n) \log n)$
3) *Se $\exists \epsilon > 0 : f(n) = \Omega(n^{\log_b a + \epsilon}) \wedge \exists c : 0 < c < 1, \exists m \geq 0 : af(n/b) \leq cf(n) \forall n \geq m$*
   allora $T(n) = \Theta(f(n))$

TEO.3 Ricorrenze lineari di ordine costante

Siano $a_1, a_2, ..., a_h$ costanti intere non negative con h costante e positivo. Siano poi c e β costanti reali tali che $c > 0$ e $\beta \geq 0$. Sia infine $T(n)$ definita dalla seguente funzione di ricorrenza:

$$T(n) = \begin{cases} \sum_{i=1}^{h} (a_i T(n-i)) + cn^{\beta} & \text{se} \quad n > m \\ \Theta(1) & \text{se} \quad n \leq m \leq h \end{cases} \tag{4}$$

Allora, posto $a = \sum_{i=1}^{h} a_i$ vale:

1) $a = 1 \implies T(n) = \Theta(n^{\beta+1})$
2) $a \geq 2 \implies T(n) = \Theta(a^n \cdot n^{\beta})$

11.3 Proprietà dei logaritmi

1. $\log_a a = 1$	*Proprietà fondamentale*
2. $\log_a 1 = 0$	*Proprietà fondamentale*
3. $\log_a b \cdot c = \log_a b + \log_a c$	*Teorema del prodotto*
4. $\log_a b \cdot c = \log_a b - \log_a c$	*Teorema del rapporto*
5. $\log_a b^c = c \cdot \log_a b$	*Teorema della potenza*
6. $\log_{a^n} bm = \frac{m \cdot \log_a b}{n}$	*Potenza alla base e all'argomento*
7. $\log_{\frac{1}{a}} b = -\log_a b$	*Base frazionaria*
8. $\log_a \frac{1}{b} = -\log_b a$	*Argomento frazionario*
9. $\log_{\frac{1}{a}} \frac{1}{b} = \log_a b$	*Base e argomento frazionario*
10. $\log_a b = \frac{1}{\log_b a}$	*Commutazione base e argomento*
11. $\log_a b = \frac{\log_c b}{\log_c a}$	*Cambio di base*
12. $a^{\log_b c} = c^{\log_b a}$	*Scambio base-argomento*

11.4 Serie matematiche convergenti

1. $\sum_{k=0}^{+\infty} k = \frac{k(k+1)}{2}$	*Formula di Gauss*
2. $\sum_{k=0}^{n} q^k = \frac{1-q^{n+1}}{1-q} = \frac{q^{n+1}-1}{q-1} \forall q : \lvert q\rvert \geq 1$	*Serie geometrica finita*
3. $\sum_{k=0}^{n} q^k = \frac{1}{1-q} \forall q : \lvert q\rvert < 1$	*Serie geometrica finita*
4. $\sum_{k=0}^{+\infty} q^k = \frac{1}{1-q} \forall \lvert q\rvert < 1$	*Serie geometrica infinita decrescente*
5. $\sum_{k=0}^{+\infty} kq^k = \frac{q}{(1-q)^2} \forall \lvert q\rvert < 1$	*Serie geometrica infinita decrescente*
6. $\sum_{k=1}^{+\infty} \frac{1}{k(k+1)} = \sum_{k=1}^{+\infty} (\frac{1}{k} - \frac{1}{k+1}) = 1$	*Serie di Mengoli*