Algoritmi

Indice degli algoritmi

- Algoritmi basati su confronto:
  - MergeSort
  - BubbleSort
  - InsertionSort
  - SelectionSort
  - QuickSort
  - HeapSort
- Algoritmi non basati su confronto
  - IntegerSort
  - BucketSort
  - RadixSort
- Algoritmi di visita di un albero
  - Algoritmo DFS
  - Algoritmo BFS
- Algoritmi di visita di un grafo
  - Algoritmo DFS
  - Algoritmo BFS

Per una spiegazione più dettagliata di qeusti algoritmi si rimanda a questo link Algoritmi

Algoritmi basati su confronto

BubbleSort

Pseudo-codice

Pseudo-codice:

Algorithm 1 BubbleSort

```
Require: array A
  while scambio = True do
    scambio = False
    for i = 0 to n − 2 do
      if A[i] > A[i + 1] then
        swap(A[i], A[i + 1])
        scambio = True
      end if
    end for
  end while
```

n è la lunghezza dell'array

Complessità temporale

l'Upper bound dell'algoritmo è:

T(n)=#passi elementari eseguiti su una RAM
c_j= costo della j-esima riga

- linee 1,3,5,6,7 hanno costo costante
- linea 2 eseguita al più n volte
- linea 4 eseguita al più n-2 volte per ogni ciclo esterno

$$T(n) \leq c_1 + nc_2 + n(n-2)(c_3 + c_4 + c_5) \implies T(n) = O(n^2) \implies T(n) = \Theta(n^2)$$

InsertionSort

Spiegazione dettagliata qui -> Algoritmi,lezione 13/10/21

Pseudo-codice

InsertionSort2 (array A)

```
1.   for k=1 to n-1 do
2.       x = A[k+1]
3.       j = k
4.       while j > 0 e A[j] > x do
5.           A[j+1] = A[j]
6.           j= j-1
7.       A[j+1]=x
```

Complessità temporale

- linea 1 eseguita n-2 volte
- linea 2,3 costo costante,eseguite al più n volte
- linea 4 eseguita al più n-2 volte (In realtà sarebbe $\sum_{j=2}^{n} t_j$) per ciclo esterno
- linee 5,6 costo costante, eseguite al più n-2 volte (In realtà sarebbe $\sum_{j=2}^{n} t_j$)
- linea 7 costo costante
  t_j è il numero di volte che la linea 4 viene eseguita

$T(n) \leq n((c_2 + c_3 + c_7 + c_6 + c_5) + (n-2)) = nc + n^2 \implies$
$T(n) = O(n^2) \implies T(n) = \Theta(n^2)$

HeapSort

Spiegazione qui -> HeapSort

Pseudo-codice

Pseudo-codice HeapSort:

```
heapSort (A)
    Heapify(A)
    Heapsize[A]=n
    for i=n down to 2 do
        scambia A[1] e A[i]
        Heapsize[A] = Heapsize[A] -1
        fixHeap(1,A)
```

Pseudo-codice fixHeap(1,A)

```
fixHeap (i,A)
    s=sin(i)
    d=des(i)
    if (s ≤ heapsize[A] e A[s] >A[i])
    then massimo=s
    else massimo=i
    if (d ≤ heapsize[A] e A[d] >A[massimo])
    then massimo=d
    if (massimo≠i)
    then scambia A[i] e A[massimo]
        fixHeap(massimo,A)
```

Pseudo-codice heapify

```
heapify-posizionale-iterativo(array A)
    n ← lunghezza di A
     for i=⌊n/2⌋ down to 1 do
          fixHeap-posizionale(i, A)
```

Complessità temporale

Per complessità temporale di heapify si rimanda alla lezione [Lezione 6 - Complessità heapify]

Complessità HeapSort:

- linea 1 costo $O(n)$ (costruzione dell'heap)
- linea 3-6 esegue n-1 estrazioni di costo $O(log(n))$

Quindi

$$T(n) \leq (n-1)log(n) \implies O(nlog(n))$$

MergeSort

Spiegazione qui -> MergeSort

Pseudo-codice

Pseudo-codice procedura Merge

Merge (A, i_1, f_1, f_2)

Sia X un array ausiliario di lunghezza f_2-i_1+1

$i=1$; $k_1=i_1$

$k_2=f_1+1$

while $(k_1 \leq f_1$ e $k_2 \leq f_2)$ **do**

    if $(A[k_1] \leq A[k_2])$

    then $X[i]=A[k_1]$

        incrementa i e k_1

    else $X[i]=A[k_2]$

        incrementa i e k_2

if $(k_1 \leq f_1)$ **then** copia $A[k_1;f_1]$ alla fine di X

else copia $A[k_2;f_2]$ alla fine di X

copia X in $A[i_1;f_2]$

Pseudo-codice MergeSort

MergeSort (A, i, f)

if $(i < f)$ **then**

    $m = \lfloor (i+f)/2 \rfloor$

    MergeSort(A,i,m)

    MergeSort(A,m+1,f)

    Merge(A,i,m,f)

Complessità temporale

La complessità termporale del MergeSort è descritto dalla seguente relazione di ricorrenza:

$$T(n) = 2T(n/2) + O(n)$$

Usando il Teorema Master abbiamo che:

$$T(n) = O(nlog(n))$$

SelectionSort

Spiegazione qui -> SelectionSort

Pseudo-codice

```
SelectionSort (array A)
    for k=1 to n-1 do
        m = k
        for j=k+1 to n do
            if (A[j] < A[m]) then m=j
        scambia A[m] con A[k]
```

Complessità temporale

Upper Bound:

$$T(n) \leq 5n^2 O(1) = \Theta(n^2) \implies T(n) = O(n^2)$$

Lower Bound:

$$T(n) \geq \sum_{k=0}^{n-2}(n-k-1) = \sum_{k=1}^{n-1}(k) = n(n-1)/2 = \Theta(n^2) \implies T(n) = \Omega(n^2)$$

Upper Bound $O(n^2)$ e Lower Bound $\Omega(n^2)$ allora $T(n) = \Theta(n^2)$

QuickSort

Spiegazione qui -> QuickSort

Pseudo-codice

Pseudo-codice Partition

```
Partition (A, i, f )
    x=A[i]
    inf =i
    sup= f + 1
    while (true) do
        do (inf=inf + 1) while (inf ≤ f e A[inf] ≤ x)
        do (sup=sup-1) while (A[sup] > x)
        if (inf < sup) then scambia A[inf] e A[sup]
        else break
    scambia A[i] e A[sup]
    return sup
```

Pseudo-codice QuickSort

```
QuickSort (A, i, f )
    if (i < f) then
        m=Partition(A,i,f)
        QuickSort(A,i,m-1)
        QuickSort(A, m +1,f)
```

Complessità temporale

Upper Bound:

$$T(n) = T(n-1) + T(0) + O(n) = T(n-1) + O(1) + O(n) = T(n-1) + O(n) \implies T(n) = O(n^2)$$

Lower Bound:

$$T(n) = \Omega(nlog(n))$$

Algoritmi non basati su confronto

IntegerSort

Pseudo-codice

```
IntegerSort (X, k)
1.    Sia Y un array di dimensione k
2.    for i=1 to k do Y[i]=0
3.    for i=1 to n do incrementa Y[X[i]]
4.    j=1
5.    for i=1 to k do
6.        while (Y[i] > 0) do
7.            X[j]=i
8.            incrementa j
9.            decrementa Y[i]
```

Complessità temporale

- linea 1 costo $O(1)$
- linea 2 costo $O(k)$
- linea 3 costo $O(n)$
- linea 4 costo $O(1)$
- linea 5 costo $O(k)$
- linee 6-9, per i fissato il num. di volte eseguite è al più $1 + Y[i] \implies O(k+n)$

Quindi:

$$T(n) \leq \sum_{i=1}^{k}(1 + Y[i]) = \sum_{i=1}^{k} 1 + \sum_{i=1}^{k}(Y[i]) = k + n \implies O(n+k)$$

Analisi

- Tempo $O(1) + O(k) = O(k)$ per inizializzare Y a 0
- Tempo $O(1) + O(n) = O(n)$ per calcolare i valori dei contatori
- Tempo $O(n+k)$ per ricostruire X
  $O(n+k)$

Tempo linearea se $k = O(n)$

BucketSort

Spiegazione qui -> [BucketSort]

Pseudo-codice

Pseudo codice BucketSort

```
BucketSort (X, k)
    Sia Y un array di dimensione k
    for i=1 to k do Y[i]=lista vuota
    for i=1 to n do
        appendi il record X[i] alla lista Y[chiave(X[i])]
    for i=1 to k do
        copia ordinatamente in X gli elemeti della lista Y[i]
```

Complessità temporale

Come l'IntegerSort, quindi $O(n + k)$

RadixSort

Spiegazione qui -> [RadixSort]

Pseudo-codice

Pseudo codice RadixSort

> RadixSort(A)

Complessità

- $O(log_b k)$ passate di BucketSort
- Ciascuna passata richiede tempo $O(n + b)$
  Quindi:

$$O((n + b)log_b k)$$

Se $b = \Theta(n)$, si ha $O(nlog_n k) = O[n\frac{log(k)}{log(n)}]$

Tempo lineare se $k = O(n^c)$, c costante

Codici degli algoritmi in python

BubbleSort

Python

```python
def bubble_sort( a ):
    n = len(a)
    ordinata = False
    num_scansioni = 1
    while not ordinata:
        print(num_scansioni)
        num_scansioni += 1
        ordinata = True
        for i in range(n-1):
            if a[i] > a[i+1]:
                a[i], a[i+1] = a[i+1], a[i]
                ordinata = False
b = [9,8,7,6,5,4,3,2,1]
bubble_sort(b)
print(b)
```

MergeSort

Python

```python
def merge( a, lx, cx, rx ):
    '''
    Precondizione: a lista e a[lx:cx] e a[cx:rx] ordinate in modo non decrescente
    Modifica a fondendo le due sottoliste in modo che a[lx:rx] risulti ordinata
    Sia n = len(a), e k = rx-lx
    '''
    i, j = lx, cx # indice in a[lx:cx] ed in a[cx:rx] rispettivamente
    c = [] # lista di output
    while i < cx and j < rx:
        if a[i] < a[j]:
            c.append(a[i])
```

```python
            i += 1
        else:
            c.append(a[j])
            j += 1
    c += a[i:cx] + a[j:rx]
    for i in range(len(c)):
        a[lx+i] = c[i]

def merge_sort(a, lx, rx):
    '''
    Precondizione: a una lista numerica
    Ordina a[lx:rx]
    '''
    if lx <= rx-2: # almeno due elementi in a[lx:rx]
        cx = (rx+lx)//2
        merge_sort(a, lx, cx)
        merge_sort(a, cx, rx)
        a = merge(a, lx, cx, rx)
a = [2,1,10,5,7,0,4,9,6,8,11,1,2]
n = len(a)
merge_sort(a, 0, n)
print(a)
```

IntegerSort

```python
def IntegerSort(A,k):
    Y=[0]*k
    n=len(A)
    for i in range(n-1):
        Y[A[i]]+=1
    j=0
    for i in range(k):
        while(Y[i]>0):
            A[j]=i
            j+=1
            Y[i]-=1
    return A
a = [1,10,4,3,3,5,20]
print(IntegerSort(a,20))
```

QuickSort

```python
def QuickSort(A,i,f):
    if(i<f):
        m=Partition(A,i,f)
        QuickSort(A,i,m-1)
        QuickSort(A,m+1,f)
    return A

def Partition(A,i,f):
    x=A[i]
    inf=i
    sup=f+1
    while True:
        inf=inf+1
        while inf<= f and A[inf]<= x:
            inf=inf+1
        sup=sup-1
        while A[sup]>x:
            sup=sup-1
        if inf< sup:
            A[inf],A[sup]=A[sup],A[inf]
        else:
            break
    A[i],A[sup]=A[sup],A[i]
    return sup

a = [1,10,4,3,3,5,20]
print(QuickSort(a,0,6))
```

SelectionSort

```python
def SelectionSort(a):
    n=len(a)
    for k in range(n-2):
        m=k+1
        for j in range(k+2,n):
            if a[j]<a[m]:
```

```python
                m=j
        a[m],a[k+1]=a[k+1],a[m]
    return a
a = [1,10,4,3,3,5,20]
print(SelectionSort(a))
```

HeapSort

da completare

Python

```python
import math
def HeapSort(a):
    heapify(a)
    heapsize = len(a)-1
    for i in range(heapsize,2,-1):
        a[1],a[i]=a[i],a[1]
        heapsize -=1
        fixHeap(1,a)
    return a

def fixHeap(i,a):
    heapsize = len(a)-1
    sx = 2*i
    dx = 2*i+1
    if sx<=heapsize and a[sx]>a[i]:
        max = sx
    else:
        max = i
    if dx<=heapsize and a[dx]>a[max]:
        max = dx
    if max != i:
        a[i],a[max] = a[max],a[i]
        fixHeap(max,a)

def heapify(a):
    heapsize = len(a)-1
    n=heapsize
    for i in range(math.floor(n/2),1):
        fixHeap(i,a)

```

```python
a = [1,10,4,3,3,5,20]
print(HeapSort(a))
```

Algoritmi di visita su un albero

Algoritmo DFS

Pseudo-codice

```
algoritmo visitaDFS(nodo r)
    Pila S
    S.push(r)
    while (not S.isEmpty()) do
        u ←S.pop()
        if (u ≠ null) then
            visita il nodo u
            S.push(figlio destro di u)
            S.push(figlio sinistro di u)
```

Complessità

```
algoritmo visitaDFS(nodo r)
    Pila S
    S.push(r)
    while (not S.isEmpty()) do
        u ←S.pop()
        if (u ≠ null) then
            visita il nodo u
            S.push(figlio destro di u)
            S.push(figlio sinistro di u)
```

Ogni nodo inserito e estratto dalla Pila una sola volta

Tempo speso per ogni nodo: O(1)
(se so individuare i figli di un nodo in tempo costante)

nodi null inseriti/estratti: O(n)

Quindi $T(n) = O(n)$

Codice in python

Python

```python
class TreeNode:
    def __init__(self, val, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def DFS(root):
    S = []
    S.append(root)
    while len(S) > 0:
        u = S.pop()
        if u != None:
            print(u.val)
            S.append(u.right)
            S.append(u.left)

root = TreeNode("A")
l1 = TreeNode("L")
l2 = TreeNode("E")
r1 = TreeNode("R")
r2 = TreeNode("B")
r3 = TreeNode("O")

root.left = l1
root.right = r2
l1.left = l2
l1.right = r1
r2.right = r3
DFS(root)
```

Versione ricorsiva, con Visita in preordine,postordine,simmetrica

Python

```python
def DFS_postorder(root):
    if root != None:
        DFS_postorder(root.left)
        DFS_postorder(root.right)

```

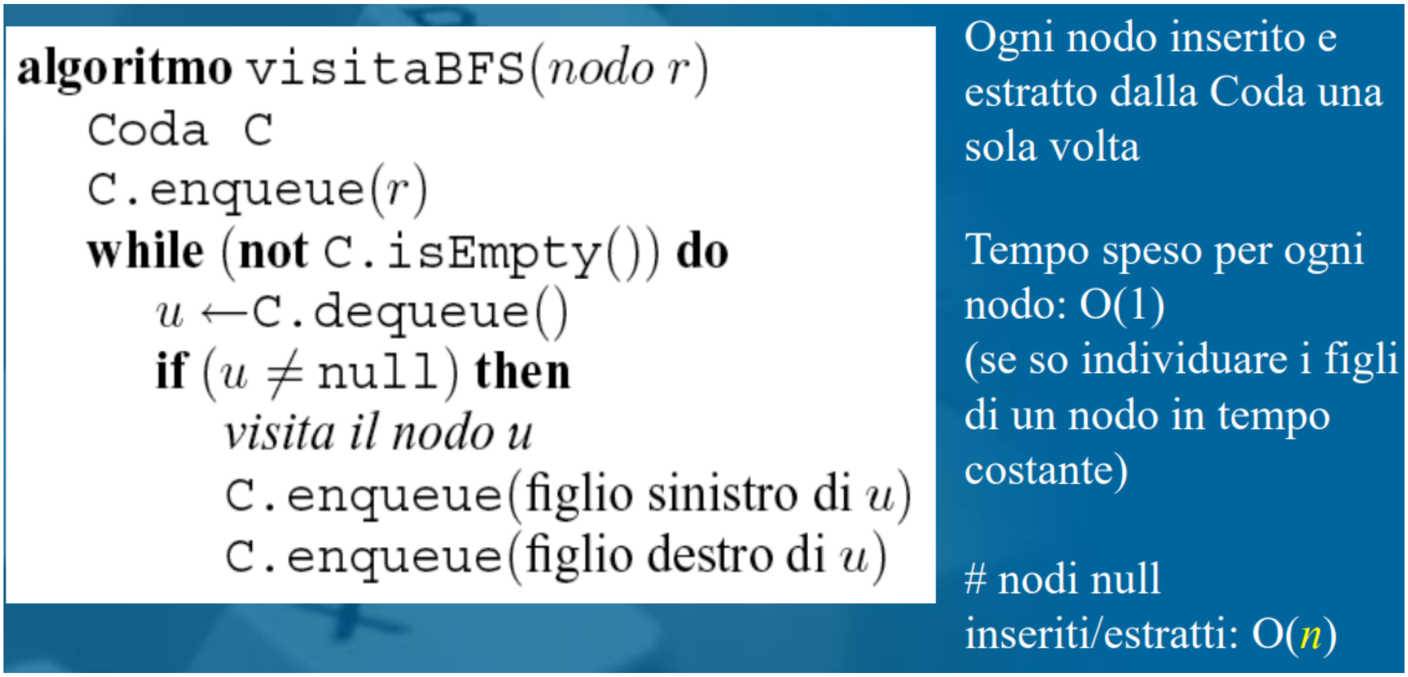
```python
        print(root.val)

def DFS_sim(root):
    if root != None:
        DFS_sim(root.left)
        print(root.val)
        DFS_sim(root.right)

def DFS_preorder(root):
    if root != None:
        print(root.val)
        DFS_preorder(root.left)
        DFS_preorder(root.right)
```

Algoritmo BFS

Pseudo-codice

```
algoritmo visitaBFS(nodo r)
    Coda C
    C.enqueue(r)
    while (not C.isEmpty()) do
        u ←C.dequeue()
        if (u ≠ null) then
            visita il nodo u
            C.enqueue(figlio sinistro di u)
            C.enqueue(figlio destro di u)
```

Complessità temporale



Quindi $T(n) = O(n)$

Codice in python

Python

```python
class TreeNode:
    def __init__(self, val, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Queue:
    def __init__(self):
        self.items = []
    def isEmpty(self):
        return self.items == []
    def enqueue(self, item):
        self.items.insert(0,item)
    def dequeue(self):
        return self.items.pop()
    def size(self):
        return len(self.items)

def BFS(root):
    c = Queue()
    c.enqueue(root)
```

```python
    while not c.isEmpty():
        u = c.dequeue()
        if u != None:
            print(u.val)#visito il nodo u
            c.enqueue(u.right)
            c.enqueue(u.left)

root = TreeNode("A")
l1 = TreeNode("L")
r1 = TreeNode("B")
l2 = TreeNode("E")
r2 = TreeNode("R")
r3 = TreeNode("O")

root.left = l1
l1.right = r1
r1.left = l2
l2.right = r2
r2.right = r3

BFS(root)
```

Algoritmi di visita di un grafo

Algoritmo DFS (Grafi)

Speigazione qua -> [Lezione 14 - DFS]

Pseudocodice

```
procedura visitaDFSRicorsiva(vertice v, albero T)
    marca e visita il vertice v
    for each ( arco (v, w) ) do
        if ( w non è marcato ) then
            aggiungi l'arco (v, w) all'albero T
            visitaDFSRicorsiva(w, T)

algoritmo visitaDFS(vertice s) → albero
    T ← albero vuoto
    visitaDFSRicorsiva(s, T)
    return T
```

Costo computazionale

Il tempo di esecuzione dipende dalla struttura dati usata per rappresentare il grafo (e dalla connettività o meno del grafo rispetto ad s):

- Liste di adiacenza: $O(m+n)$
- Matrice di adiacenza: $O(n^2)$

Algoritmo BFS (Grafi)

Speigazione qua -> [Lezione 14 - BFS]

Pseudocodice

```
algoritmo visitaBFS(vertice s) → albero
    rendi tutti i vertici non marcati
    T ← albero formato da un solo nodo s
    Coda F
    marca il vertice s
    F.enqueue(s)
    while ( not F.isempty() ) do
        u ← F.dequeue()
        for each ( arco (u, v) in G ) do
            if ( v non è ancora marcato ) then
                F.enqueue(v)
                marca il vertice v
                rendi u padre di v in T
    return T
```

Costo Computazionale

Il tempo di esecuzione dipende dalla sruttura dati usata per rappresentare il grafo (e dalla connettività o meno del grafo rispetto ad s):

- Liste di adiacenza: $O(m + n)$
- Matrice di adiacenza: $O(n^2)$