

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 questi 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)

1. Heapify(A)
2. Heapsize[A]=n
3. **for** i=n **down to** 2 **do**
4. scambia A[1] e A[i]
5. Heapsize[A] = Heapsize[A] -1
6. fixHeap(1,A)

Pseudo-codice fixHeap(1,A)

fixHeap (i,A)

1. s=sin(i)
2. d=des(i)
3. **if** ($s \leq \text{heapsize}[A]$ e $A[s] > A[i]$)
4. **then** massimo=s
5. **else** massimo=i
6. **if** ($d \leq \text{heapsize}[A]$ e $A[d] > A[\text{massimo}]$)
7. **then** massimo=d
8. **if** (massimo≠i)
9. **then** scambia A[i] e A[massimo]
10. fixHeap(massimo,A)

Pseudo-codice heapify

heapify-posizionale-iterativo(array A)

1. $n \leftarrow \text{lunghezza di } A$
2. **for** $i = \lfloor n/2 \rfloor$ **down to** 1 **do**
3. **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

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

MergeSort

Spiegazione qui -> [MergeSort](#)

Pseudo-codice

Pseudo-codice procedura Merge

Merge (A, i_1, f_1, f_2)

1. Sia X un array ausiliario di lunghezza $f_2 - i_1 + 1$
2. $i = 1; k_1 = i_1$
3. $k_2 = f_1 + 1$
4. **while** ($k_1 \leq f_1$ e $k_2 \leq f_2$) **do**
5. **if** ($A[k_1] \leq A[k_2]$)
6. **then** $X[i] = A[k_1]$
7. incrementa i e k_1
8. **else** $X[i] = A[k_2]$
9. incrementa i e k_2
10. **if** ($k_1 \leq f_1$) **then** copia $A[k_1; f_1]$ alla fine di X
11. **else** copia $A[k_2; f_2]$ alla fine di X
12. copia X in $A[i_1; f_2]$

Pseudo-codice MergeSort

MergeSort (A, i, f)

1. **if** ($i < f$) **then**
2. $m = \lfloor (i+f)/2 \rfloor$
3. MergeSort(A, i, m)
4. MergeSort($A, m+1, f$)
5. Merge(A, i, m, f)

Complessità temporale

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

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

Usando il Teorema Master abbiamo che:

$$T(n) = O(n \log(n))$$

SelectionSort

Spiegazione qui -> [SelectionSort](#)

Pseudo-codice

SelectionSort (array A)

1. **for** k=1 **to** n-1 **do**
2. m = k
3. **for** j=k+1 **to** n **do**
4. **if** (A[j] < A[m]) **then** m=j
5. 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)

1. $x = A[i]$
2. $inf = i$
3. $sup = f + 1$
4. **while** (true) **do**
5. **do** ($inf = inf + 1$) **while** ($inf \leq f \wedge A[inf] \leq x$)
6. **do** ($sup = sup - 1$) **while** ($A[sup] > x$)
7. **if** ($inf < sup$) **then** scambia $A[inf]$ e $A[sup]$
8. **else break**
9. scambia $A[i]$ e $A[sup]$
10. **return** sup

Pseudo-codice QuickSort

QuickSort (A, i, f)

1. **if** ($i < f$) **then**
2. $m = \text{Partition}(A, i, f)$
3. QuickSort(A, i, m-1)
4. 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:

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

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 lineare se $k = O(n)$

BucketSort

Spiegazione qui -> [BucketSort](#)

Pseudo-codice

Pseudo codice BucketSort

```
BucketSort (X, k)
1.   Sia Y un array di dimensione k
2.   for i=1 to k do Y[i]=lista vuota
3.   for i=1 to n do
4.       appendi il record X[i] alla lista Y[chiave(X[i])]
5.   for i=1 to k do
6.       copia ordinatamente in X gli elementi 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 $O(n \log_n k) = O\left[n \frac{\log(k)}{\log(n)}\right]$

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

Codici degli algoritmi in python

BubbleSort

Python ▾

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

MergeSort

Python ▾

```
1  def merge( a, lx, cx, rx ):
2      '''
3      Precondizione: a lista e a[lx:cx] e a[cx:rx] ordinate in
4      modo non decrescente
5      Modifica a fondendo le due sottoliste in modo che a[lx:rx]
6      risulti ordinata
7      Sia n = len(a), e k = rx-lx
8      '''
9      i, j = lx, cx # indice in a[lx:cx] ed in a[cx:rx]
10     rispettivamente
11     c = [] # lista di output
12     while i < cx and j < rx:
13         if a[i] < a[j]:
14             c.append(a[i])
15
```

```

15         i += 1
16     else:
17         c.append(a[j])
18         j += 1
19     c += a[i:cx] + a[j:rx]
20     for i in range(len(c)):
21         a[lx+i] = c[i]
22
23 def merge_sort(a, lx, rx):
24     '''
25     Precondizione: a una lista numerica
26     Ordina a[lx:rx]
27     '''
28     if lx <= rx-2: # almeno due elementi in a[lx:rx]
29         cx = (rx+lx)//2
30         merge_sort(a, lx, cx)
31         merge_sort(a, cx, rx)
32         a = merge(a, lx, cx, rx)
33
34 a = [2,1,10,5,7,0,4,9,6,8,11,1,2]
35 n = len(a)
36 merge_sort(a, 0, n)
37 print(a)

```

IntegerSort

Python ▾

```

1 def IntegerSort(A,k):
2     Y=[0]*k
3     n=len(A)
4     for i in range(n-1):
5         Y[A[i]]+=1
6     j=0
7     for i in range(k):
8         while(Y[i]>0):
9             A[j]=i
10            j+=1
11            Y[i]-=1
12     return A
13 a = [1,10,4,3,3,5,20]
14 print(IntegerSort(a,20))

```

QuickSort

Python ▾

```
1 def QuickSort(A,i,f):
2     if(i<f):
3         m=Partition(A,i,f)
4         QuickSort(A,i,m-1)
5         QuickSort(A,m+1,f)
6     return A
7
8 def Partition(A,i,f):
9     x=A[i]
10    inf=i
11    sup=f+1
12    while True:
13        inf=inf+1
14        while inf<= f and A[inf]<= x:
15            inf=inf+1
16        sup=sup-1
17        while A[sup]>x:
18            sup=sup-1
19        if inf< sup:
20            A[inf],A[sup]=A[sup],A[inf]
21        else:
22            break
23    A[i],A[sup]=A[sup],A[i]
24    return sup
25
26 a = [1,10,4,3,3,5,20]
27 print(QuickSort(a,0,6))
```

SelectionSort

Python ▾

```
1 def SelectionSort(a):
2     n=len(a)
3     for k in range(n-2):
4         m=k+1
5         for j in range(k+2,n):
6             if a[j]<a[m]:
7                 -
```

```

7             m=j
8             a[m],a[k+1]=a[k+1],a[m]
9         return a
10    a = [1,10,4,3,3,5,20]
11    print(SelectionSort(a))

```

HeapSort

da completare

Python ▾

```

1  import math
2  def HeapSort(a):
3      heapify(a)
4      heapsize = len(a)-1
5      for i in range(heapsize,2,-1):
6          a[1],a[i]=a[i],a[1]
7          heapsize -=1
8          fixHeap(1,a)
9      return a
10
11 def fixHeap(i,a):
12     heapsize = len(a)-1
13     sx = 2*i
14     dx = 2*i+1
15     if sx<=heapsize and a[sx]>a[i]:
16         max = sx
17     else:
18         max = i
19     if dx<=heapsize and a[dx]>a[max]:
20         max = dx
21     if max != i:
22         a[i],a[max] = a[max],a[i]
23         fixHeap(max,a)
24
25 def heapify(a):
26     heapsize = len(a)-1
27     n=heapsize
28     for i in range(math.floor(n/2),1):
29         fixHeap(i,a)
30

```

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

Insertion Sort

Python ▾

```
1 def InsertionSort(a):
2     n=len(a)
3     for k in range(n-1):
4         x = a[k+1]
5         j=k
6         while j>0 and a[j]>x:
7             a[j+1] = a[j]
8             j=j-1
9         a[j]=x
10    return a
11
12 a = [8,4,6,3,1,1,5,2,4,8,6,2,0,4,7,9]
13 a_s = InsertionSort(a)
14 print(a_s)
```

Algoritmi di visita su un albero

Algoritmo DFS

Pseudo-codice

```
algoritmo visitaDFS(nodo  $r$ )
    Pila  $S$ 
     $S.push(r)$ 
    while (not  $S.isEmpty()$ ) do
         $u \leftarrow S.pop()$ 
        if ( $u \neq null$ ) then
            visita il nodo  $u$ 
             $S.push(\text{figlio destro di } u)$ 
             $S.push(\text{figlio sinistro di } u)$ 
```

Complessità

```
algoritmo visitaDFS(nodo  $r$ )  
  Pila  $S$   
   $S.push(r)$   
  while (not  $S.isEmpty()$ ) do  
     $u \leftarrow S.pop()$   
    if ( $u \neq \text{null}$ ) then  
      visita il nodo  $u$   
       $S.push(\text{figlio destro di } u)$   
       $S.push(\text{figlio sinistro di } u)$ 
```

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 ▾

```
1  class TreeNode:  
2      def __init__(self, val, left=None, right=None):  
3          self.val = val  
4          self.left = left  
5          self.right = right  
6  
7  def DFS(root):  
8      S = []  
9      S.append(root)  
10     while len(S) > 0:  
11         u = S.pop()  
12         if u != None:  
13             print(u.val)  
14             S.append(u.right)  
15             S.append(u.left)  
16  
17  root = TreeNode("A")  
18  l1 = TreeNode("L")  
19  l2 = TreeNode("E")  
20
```



```

21  r1 = TreeNode("R")
22  r2 = TreeNode("B")
23  r3 = TreeNode("O")
24
25  root.left = l1
26  root.right = r2
27  l1.left = l2
28  l1.right = r1
29  r2.right = r3
    DFS(root)

```

Versione ricorsiva, con Visita in preordine,postordine,simmetrica

Python ▾

```

1  def DFS_postorder(root):
2      if root != None:
3          DFS_postorder(root.left)
4          DFS_postorder(root.right)
5          print(root.val)
6
7  def DFS_sim(root):
8      if root != None:
9          DFS_sim(root.left)
10         print(root.val)
11         DFS_sim(root.right)
12
13  def DFS_preorder(root):
14      if root != None:
15          print(root.val)
16          DFS_preorder(root.left)
17          DFS_preorder(root.right)

```

Algoritmo BFS

Pseudo-codice

```
algoritmo visitaBFS(nodo  $r$ )
  Coda  $C$ 
   $C.enqueue(r)$ 
  while (not  $C.isEmpty()$ ) do
     $u \leftarrow C.dequeue()$ 
    if ( $u \neq null$ ) then
      visita il nodo  $u$ 
       $C.enqueue(\text{figlio sinistro di } u)$ 
       $C.enqueue(\text{figlio destro di } u)$ 
```

Complessità temporale

```
algoritmo visitaBFS(nodo  $r$ )
  Coda  $C$ 
   $C.enqueue(r)$ 
  while (not  $C.isEmpty()$ ) do
     $u \leftarrow C.dequeue()$ 
    if ( $u \neq null$ ) then
      visita il nodo  $u$ 
       $C.enqueue(\text{figlio sinistro di } u)$ 
       $C.enqueue(\text{figlio destro di } u)$ 
```

Ogni nodo inserito e
estratto dalla Coda 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

```
1 class TreeNode:
2     def __init__(self, val, left=None, right=None):
3         self.val = val
4         self.left = left
```

Python ▾

```

5         self.right = right
6
7     class Queue:
8         def __init__(self):
9             self.items = []
10        def isEmpty(self):
11            return self.items == []
12        def enqueue(self, item):
13            self.items.insert(0,item)
14        def dequeue(self):
15            return self.items.pop()
16        def size(self):
17            return len(self.items)
18
19    def BFS(root):
20        c = Queue()
21        c.enqueue(root)
22        while not c.isEmpty():
23            u = c.dequeue()
24            if u != None:
25                print(u.val)#visito il nodo u
26                c.enqueue(u.right)
27                c.enqueue(u.left)
28
29    root = TreeNode("A")
30    l1 = TreeNode("L")
31    r1 = TreeNode("B")
32    l2 = TreeNode("E")
33    r2 = TreeNode("R")
34    r3 = TreeNode("O")
35
36    root.left = l1
37    l1.right = r1
38    r1.left = l2
39    l2.right = r2
40    r2.right = r3
41
42    BFS(root)

```

Algoritmi di visita di un grafo

Algoritmo DFS (Grafì)

Speigazione qua -> [Lezione 14 - DFS](#)

Pseudocodice

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

```
algoritmo visitaDFS(vertice  $s$ )  $\rightarrow$  albero  
6.    $T \leftarrow$  albero vuoto  
7.   visitaDFSRicorsiva( $s, T$ )  
8.   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 (Grafì)

Speigazione qua -> [Lezione 14 - BFS](#)

Pseudocodice

algoritmo visitaBFS(*vertice* s) \rightarrow *albero*

1. rendi tutti i vertici non marcati
2. $T \leftarrow$ albero formato da un solo nodo s
3. **Coda** F
4. marca il vertice s
5. $F.\text{enqueue}(s)$
6. **while** (**not** $F.\text{isempty}()$) **do**
7. $u \leftarrow F.\text{dequeue}()$
8. **for each** (arco (u, v) in G) **do**
9. **if** (v non è ancora marcato) **then**
10. $F.\text{enqueue}(v)$
11. marca il vertice v
12. rendi u padre di v in T
13. **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: $O(n^2)$