Corso di laurea in Informatica
Algoritmi 1
A.A. 2025/2026

Esercizi riepilogativi

Tiziana Calamoneri

SAPIENZA
UNIVERSITÀ DI ROMA

Slides realizzate sulla base di quelle preparate da T. Calamoneri e G. Bongiovanni per il corso di Informatica Generale tenuto a distanza nell'A.A. 2019/20

Esercizio svolto 1 (1)

Esercizio 1. Dato un array che contiene solo numeri negativi e positivi (nessun valore pari a zero), riorganizzarlo in modo che tutti i numeri negativi stiano a sinistra di quelli positivi.

Nota: Ordinare l'array risolve il problema, ma è inutilmente costoso. Usare il `Counting sort` può non essere possibile (non abbiamo ipotesi sul range dell'input).

IDEA 1: Basta adattare la `Partition` del `Quicksort` in modo che scambi un positivo in posizione i con un negativo in posizione j

Esercizio svolto 1 (2)

```
def SeparaPosNeg1 (A):
i,j = 0,len(A)-1
while (i < j):
    while (A[j] > 0) and (i ≤ j):
        j -= 1
    while ((A[i] < 0)and(i ≤ j):
        i += 1
    if (i < j)
     A[i], A[j]= A[j], A[i]
```

Costo computazionale: Θ(n)

Esercizio svolto 1 (3)

IDEA 2. Il problema di separare gli elementi positivi dai negativi si può ridurre a quello di ordinare n numeri interi nel range [0,1]; si può quindi applicare una modifica dell'algoritmo di Counting Sort; non serve l'array ausiliario C, perché non abbiamo dati satellite, ma serve comunque B.

```
def  SeparaPosNeg2(A):
   B=[0]*len(A)
   neg, pos = 0,len(A)-1
   for i in range len(A):
    if A[i]>0:
       B[pos] = A[i]
       pos -= 1
    else:
       B[neg] = A[i]
       neg += 1
```

Costo computazionale: $\Theta(n)$ ma spazio $\Theta(n)$

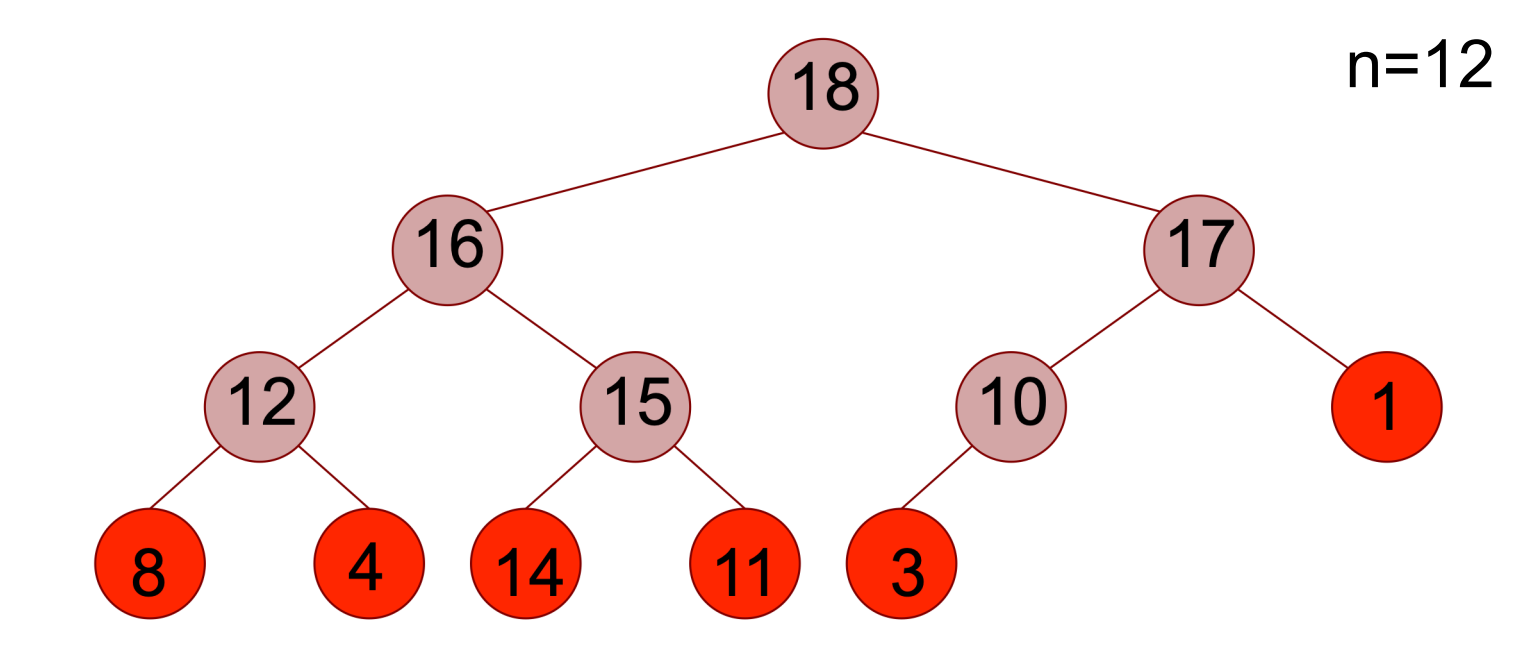
Esercizio svolto 2 (1)

Esercizio 2. Progettare un algoritmo che, dato in input un array che rappresenta uno heap, restituisca il valore minimo. Fare le opportune considerazioni sul costo computazionale.

Idea:

- Il minimo va cercato nelle foglie dello heap; anche nel caso di ripetizioni del valore minimo, una di esse deve per forza trovarsi in una foglia;
- non c'è alcuna proprietà dello heap che ci aiuti a trovare il minimo fra i valori contenuti nelle foglie

Esercizio svolto 2 (2)



18	16	17	12	15	10	1	8	4	14	11	3

Esercizio svolto 2 (3)

- sappiamo solo che le foglie stanno negli elementi dell'array che occupano le posizioni da $\lfloor n/2 \rfloor$ (compreso) in avanti;
- quindi si deve usare un ciclo che, ispezionando le posizioni dell'array da $\lfloor n/2 \rfloor$ a n-1 compresi, individui il minimo.

```
def minInHeap(A,n):
    return min([A[i] for i in range(n//2,n)])
```

Costo computazionale: Θ(n).

Esercizio svolto 3 (1)

Esercizio 3. Implementare la visita in pre-order su un albero binario memorizzato con la notazione posizionale.

IDEE

Operiamo un semplice adattamento della visita ricorsiva già vista.

La differenza è semplicemente che, anziché seguire puntatori, si calcolano gli indici dei figli (controllando che esistano).

Bisogna però anche controllare di non superare la fine dell'array (che supponiamo contenere n elementi)

Esercizio svolto 3 (2)

```
def Visita_preordine_NotPos (A,i):
  if ((i < len(A)) AND (A[i] ≠ '-')):
    %accesso al nodo e operazioni conseguenti
    Visita_preordine_NotPos (A;2*i+1)
    Visita_preordine_NotPos (A;2*i+2)
  return
```

Il costo computazionale è identico a quello della visita preorder già vista, e va dimostrato formalmente, essendo questa una funzione ricorsiva.

Esercizio svolto 4 (1)

Esercizio 4.

In un albero binario, si definisca *n*-sbilanciamento di un nodo il valore assoluto tra il numero di nodi nel suo sottoalbero sinistro ed il numero di nodi nel suo sottoalbero destro.

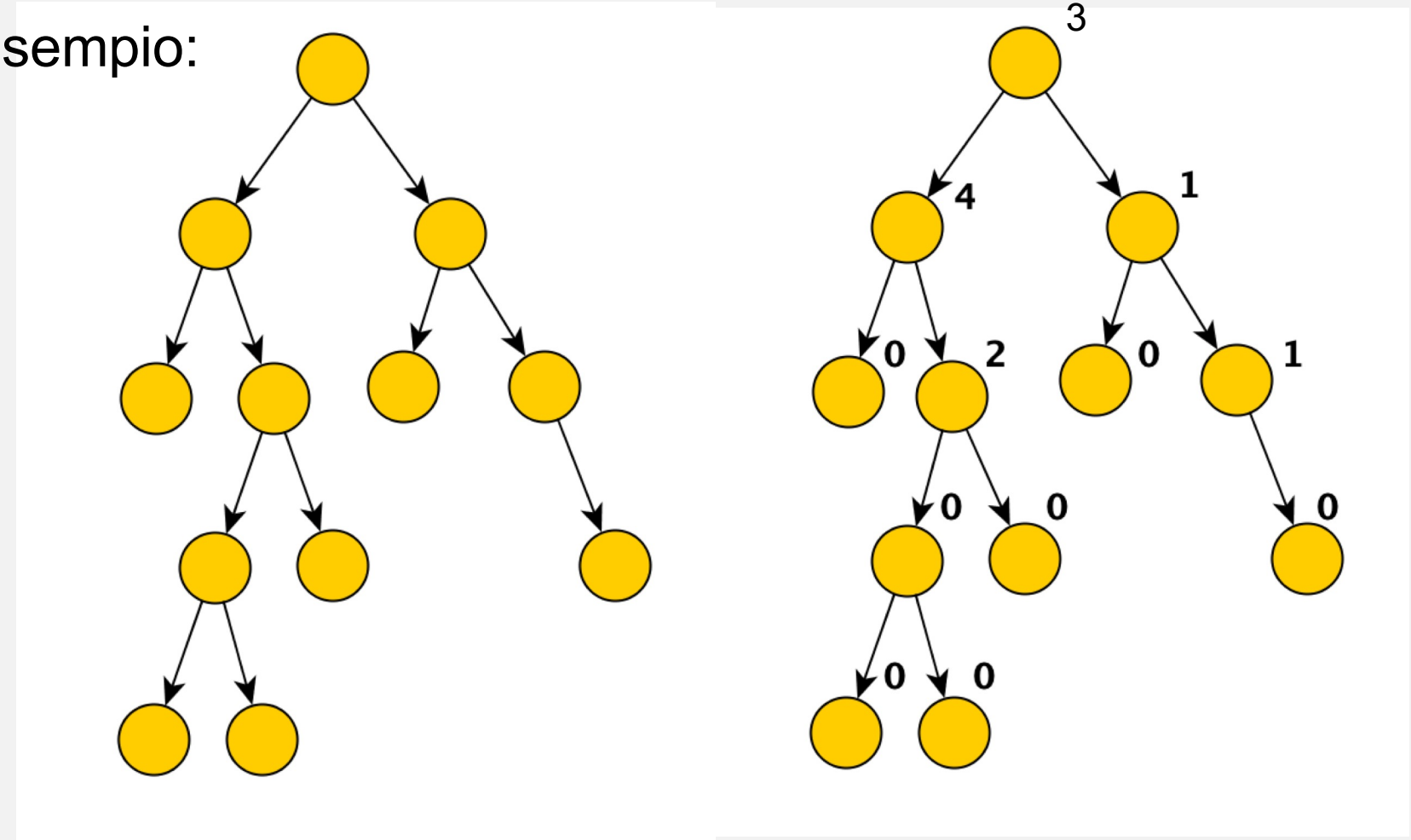
Dato un albero binario di cui si conosce il puntatore alla radice, trovare il massimo tra gli n-sbilanciamenti dei suoi nodi.

La funzione deve essere ricorsiva e NON fare uso di variabili globali.

…

Esercizio svolto 4 (2)

Esempio:



Esercizio svolto 4 (3)

IDEA

Ogni nodo riceve opportune informazioni da entrambi i suoi figli e poi effettua il calcolo seguendo la filosofia della visita in **post-ordine**.

Ogni nodo restituisce al padre sia l'n-sbilanciamento massimo per i nodi nel suo sottoalbero che il numero di nodi nel suo sottoalbero.

Al termine della visita la funzione restituirà l'n-sbilanciamento massimo (ed il numero di nodi dell'albero).

Esercizio svolto 4 (4)

```
def sbilanciamento(t):
      if (t == None):
            return 0,0
      max_sbil_sin,nodi_sin = sbilanciamento(t.left)
      max_sbil_des,nodi_des = sbilanciamento(t.right)
      massimo=max(abs(nodi_sin - nodi_des),
                              max_sbil_sin,max_sbil_des)
      return massimo,(nodi_sin + nodi_des +1)
```

Nota: La parte rossa calcola il numero di nodi dell'albero

Esercizio svolto 4 (5)

La funzione ricorsiva appena scritta, invocata su un nodo x, riceve ricorsivamente il numero di nodi ed il max n-sbilanciamento nei suoi due sottoalberi dai suoi figli, calcola il suo n-sbilanciamento ed il numero di nodi nel suo sottoalbero e restituisce tutto al padre.

Il costo computazionale è, come è ovvio, quello della visita di un albero, e l'equazione di ricorrenza relativa allo pseudocodice è:

$$T(n)=T(k)+T(n-1-k)+ \Theta(1)$$

$$T(0)= \Theta(1)$$

che si può risolvere con il metodo di sostituzione dando come soluzione $\Theta(n)$.

Esercizio svolto 5 (1)

Esercizio 5. Siano dati due array A e B, contenenti $n \geq 1$ ed $m \geq 1$ numeri reali. Gli array sono entrambi ordinati in senso crescente. A e B non contengono duplicati, ma uno stesso valore potrebbe essere presente sia in A che in B.

Progettare un algoritmo iterativo efficiente che stampi i valori che appartengono all'unione insiemistica di A e di B.

Ad esempio, se A = [1,3,4,6] e B = [2,3,4,7], l'algoritmo deve stampare 1, 2, 3, 4, 6, 7.

Determinare il costo computazionale, in funzione di n ed m.

Esercizio svolto 5 (2)

OSSERVAZIONE.

attenzione al problema dei duplicati: sappiamo che A e B contengono elementi tutti diversi tra loro ma possono esistere elementi che stanno sia in A che in B, e questi vanno stampati una volta sola.

IDEA 1.

- Considera gli elementi di A uno dopo l'altro e stampa solo quelli che non compaiono in B.
- Poi considera gli elementi di B non ancora stampati (?) uno dopo l'altro e stampali tutti.

Esercizio svolto 5 (3)

Dettaglio 1: per ogni elemento di A, scorriamo tutto B per vedere se è presente: Costo computazionale: $\Theta(nm+m)=\Theta(nm)$, ma non stiamo usando l'ipotesi che i due array siano ordinati!

Dettaglio 2: eseguiamo, per ciascun elemento di A, una ricerca binaria su B: costo computazionale $O(n \log m + m)$, ma ancora non stiamo sfruttando tutte le ipotesi perché così abbiamo usato il fatto che B sia ordinato ma l'ordinamento su A non ci serve a niente.

Esercizio svolto 5 (4)

IDEA 2.

Usiamo un algoritmo simile alla fusione di `Mergesort`, trascrivendo solo uno degli elementi uguali:

- i scorre A e j scorre B partendo entrambi da 0.
- Confrontiamo A[i] e B[j] :
  - se A[i]<B[j] si stampa A[i] e si incrementa i
  - se A[i]>B[j] si stampa B[j] e si incrementa j
  - se A[i]=B[j] si stampa A[i] e si incrementano i e j.
- Appena uno dei due array termina, stampiamo tutti gli elementi dell'altro.
- NOTA: a differenza della fusione del `MergeSort`, non abbiamo qui bisogno di un array ausiliario.

Costo computazionale: $\Theta(n+m)$.

Esercizio svolto 5 (5)

```
def Stampa_unione(A, B):
    n, m = len(A), len(B); i, j = 0, 0
    while i < n AND j <m:
        if A[i] < B[j]:
            print( A[i]); i+=1
        elif A[i] > B[j]:
            print(B[j]); j+=1
       else:
            print( A[i]); i+=1; j+=1
    while i < n: # stampa la parte finale di A
        print(A[i]); i+=1
    while j < m: # stampa la parte finale di B
        print(B[j]); j+=1
```

Il costo di questo algoritmo è $\Theta(n+m)$ in quanto effettua una scansione di entrambi gli array esaminando ciascun elemento una e una sola volta in tempo $\Theta(1)$.

Esercizio svolto 6 (1)

Esercizio 6. Sia data una funzione matematica che prende un intero, restituisce un intero, ed è strettamente crescente (vale a dire f(x)<f(x+1)).

Vogliamo trovare il primo intero non negativo per cui la funzione assume un valore non negativo.

Ad esempio, per f(x)=-100+3x il valore da trovare è 34. Progettare un algoritmo che trova questo valore in tempo O(log n), assumendo che il calcolo di f costi $\Theta(1)$.

Esercizio svolto 6 (2)

IDEA 1: Una semplice soluzione consiste nel cominciare a calcolare f(0) e, se il valore è negativo, via via incrementare x fermandosi al primo x che dà un valore non negativo.

Nel caso dell'esempio:

f(0)=-100; f(1)=-97; …; f(33)=-1; f(34)=+2.

L'algoritmo è corretto ma il suo costo computazionale è $\Theta(n)$, dove n è il valore trovato.

Esercizio svolto 6 (3)

IDEA 2: Applichiamo la ricerca binaria ma, per poterlo fare, dobbiamo ricavare un limite al range:

1. Raddoppiamo ripetutamente il valore x su cui calcolare f(x) fino a che non giungiamo ad un x' per cui f(x')≥0.
2. Applichiamo la ricerca binaria nell'intervallo [x'/2, x'] alla ricerca del min n per cui f(n) ≥0

Il costo computazionale dell'algoritmo è $\Theta(\log n)$ infatti:

1. Il passo 1. richiede tempo $\Theta(\log n)$.
2. Il passo 2 richiede tempo $O(\log n)$.

Esercizio svolto 6 (4)

```
def trovaPositivo(f) :
    if f(0) >= 0: return 0
    i = 1
    while f(i) < 0: i = i * 2
    return ricercaBinaria(i//2, i)

def ricercaBinaria(i, j):
    med = (i + j)//2
    #se f(med) è il primo non negativo nell'intervallo
    if f(med) >= 0 and (med == i or f(med-1) < 0) :
        return med
    if f(med) < 0 : # f(med) è negativo
        return ricercaBinaria(mid + 1, j)
    else : # f(med) è non negativo ma non il primo
        return ricercaBinaria(i, mid -1)

def f(x): return -100+3*x
>>> trovaPositivo(f)
34
```

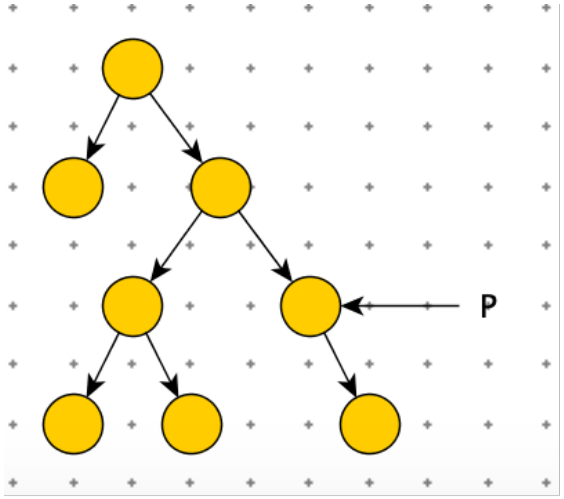
Esercizio svolto 7 (1)

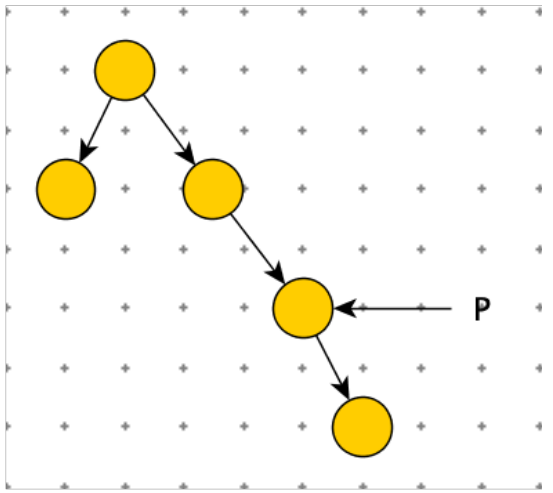
Esercizio 7. Sia dato un ABR ed un suo nodo x. Si vuole cancellare l'eventuale nodo y fratello di x e tutti i nodi nel sottoalbero radicato in y.

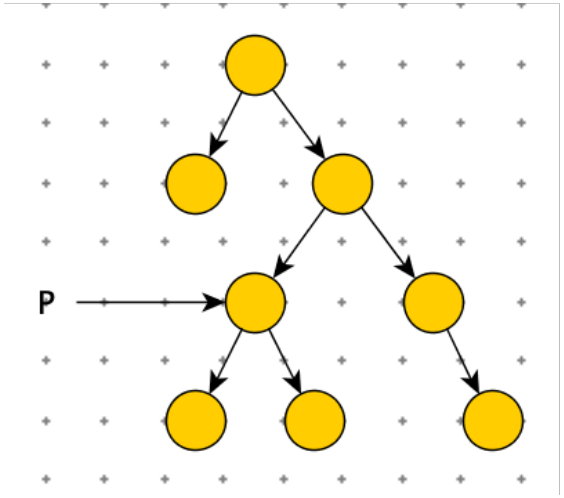
Progettare una funzione che risolva il problema quando:

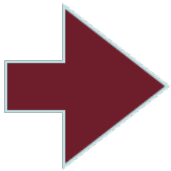
1. L'albero sia rappresentato tramite record e puntatori, dove ogni nodo abbia anche il puntatore al padre ed alla funzione venga fornito il puntatore all'albero ed il puntatore al nodo x;
2. L'albero sia rappresentato con notazione posizionale e alla funzione venga passato l'array A e l'indice i in cui si trova la chiave del nodo x.

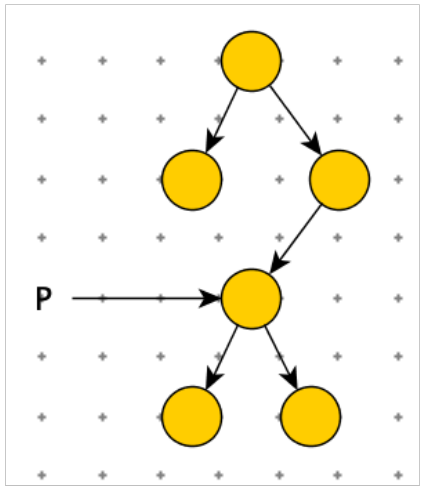
Esercizio svolto 7 (2)









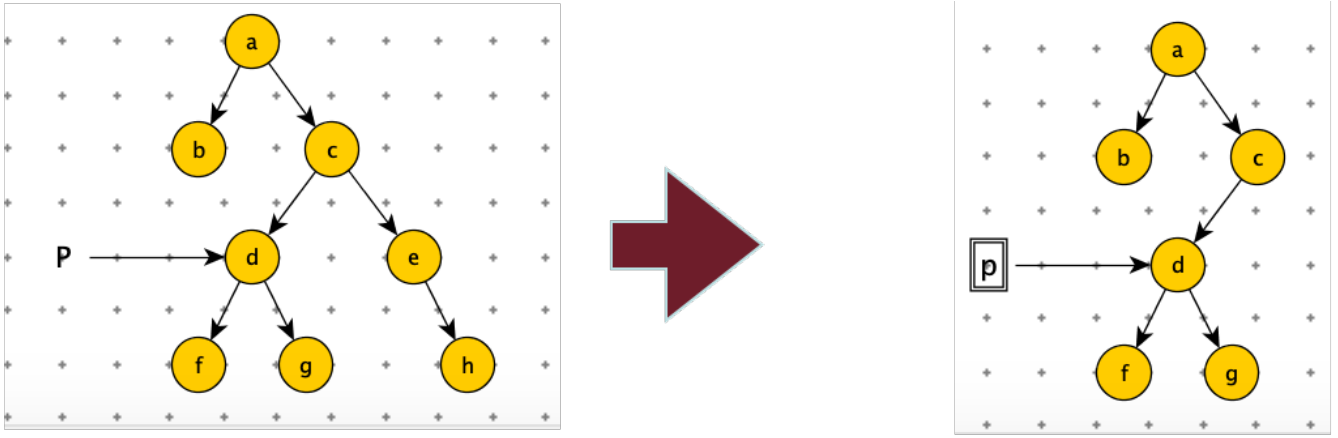


Esercizio svolto 7 (3)

```
def cancella(p):
    if p.parent=None:
          #impossibile cancellare il fratello
        return
    if p.parent.left==p: #se p è figlio des.
        p.parent.right=None
    else:                    #se p è figlio sin.
        p.parentleft=None
```

Costo: $\Theta(1)$

Esercizio svolto 7 (4)



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
a	b	c		_	d	e	-	-	-	-	f	g	-	h

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
a	b	c		_	d	-	-	-	-	-	f	-	-	-

Esercizio svolto 7 (5)

```
def cancella1(A,i):
    if i==0:
        return
    padre = (i-1)//2
    if 2*padre+1==i:
        y =2*padre+2
    else:
        y = 2*padre+1
    if y< len(A) and A[y]!=‘-‘: cancellaR(A,y)

def cancellaR(A,y):
    if 2*y+1<len(A) and A[2*y+1]!='-’:
        cancellaR(A,2*y+1)
    if 2*y+2<len(A) and A[2*y+2]!='-’:
        cancellaR(A,2*y+2)
    A[y] =‘-'
```

Costo: O(n)

Esercizio svolto 8 (1)
(esame giu. ‘21)

Esercizio 8. Data una sequenza S di n bit memorizzata in un array A, progettare un algoritmo che ordini S in tempo $\Theta(n)$.

L’algoritmo deve ordinare in loco.

Dell’algoritmo proposto si dia la descrizione a parole, si scriva lo pseudocodice e si calcoli il costo computazionale.

Per quale motivo è possibile ordinare S in tempo lineare?

Esercizio svolto 8 (2)

IDEA 1. si può usare il `Counting sort` senza dati satellite, visto che l'array A contiene solo due valori, 0 e 1.

Si ottiene un algoritmo lineare perché l'array contiene un numero costante, cioè O(n), di valori.

Esercizio svolto 8 (3)

IDEA 2. Alternativamente, è possibile implementare una leggera modifica della funzione `Partiziona` del `Quicksort`, per spostare a sinistra tutti gli zeri ed a destra tutti gli uni.

Anche in questo caso l'algoritmo è lineare come il costo di `Partiziona`.

Esercizio svolto 9 (1)
(esame lug. ‘21)

Esercizio 9. Progettare un algoritmo che, dato un array A di n interi distinti i cui elementi sono all’inizio in ordine crescente e da un certo punto in poi in ordine decrescente, restituisce in tempo O(log n) il massimo intero presente nell’array.

Ad esempio: per A = [8,10,20,80,100,200,400,500,3,2,1] l’algoritmo deve restituire il valore 500.

Esercizio svolto 9 (2)

Soluzione.

Se l'array contiene 1 o 2 elementi risolvo il problema direttamente. In caso contrario applico la ricerca binaria: sia m la posizione di mezzo dell'array, se A[m − 1] < A[m] cerco nel sottoarray di destra che parte dalla posizione m in caso contrario cerco nel sottoarray di sinistra che termina nella posizione A[m − 1].

Esercizio svolto 9 (3)

```
def esame2(A):
    i=0
    j = len(A) - 1
    while True:
        if j == i: return A[i]
        if j == i + 1: return max(A[i], A[i + 1])
        m = (i + j)//2
        if A[m-1]<A[m]:i=m
        else: j=m-1
```

Costo: O(log n)

Esercizio svolto 10 (1)

Esercizio 10. Progettare una funzione che prende la testa di una lista puntata di n nodi e restituisce il puntatore ad una nuova lista puntata che contiene i nodi con le chiavi pari presenti nella prima lista.

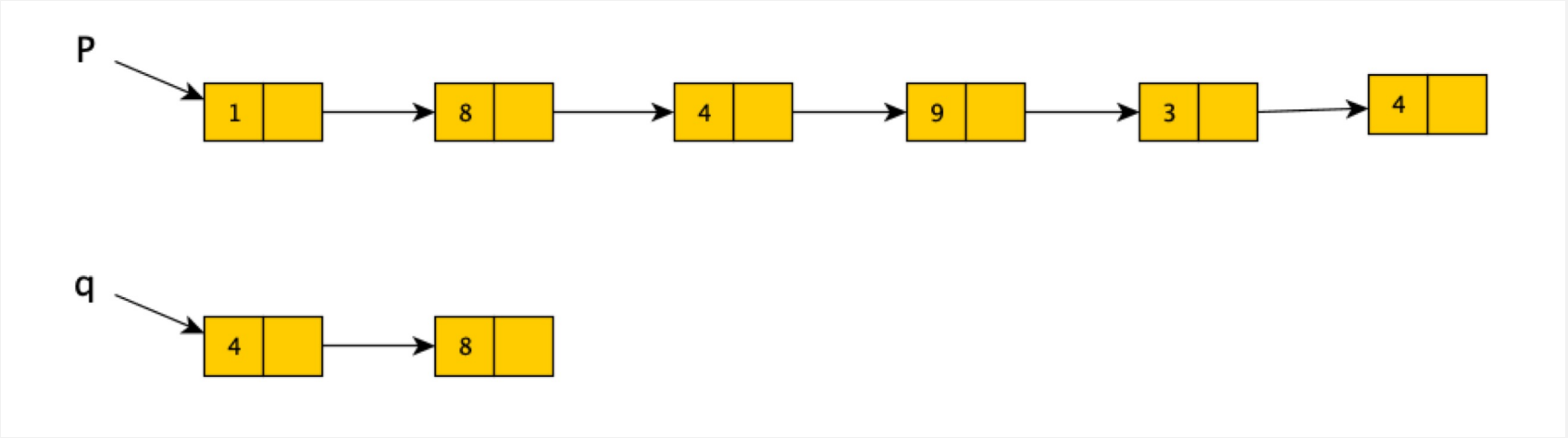
La lista originaria non va modificata, le chiavi della nuova lista devono essere ordinate in modo crescente e la lista non deve contenere duplicati.

Il costo dell'algoritmo deve essere $O(n^2)$.

…

Esercizio svolto 10 (2)

Per esempio per la lista p va restituita la lista q



Descrivere l'algoritmo, scrivere lo pseudocodice e dimostrare il costo computazionale

Esercizio svolto 10 (3)

IDEA:

- L'algoritmo parte con la lista q vuota.
- Scorre la lista p e, per ogni chiave pari x che incontra, la inserisce in q (se non già presente) in modo ordinato tramite una funzione Inserisci(q, x).

L'idea sembra corretta. Il costo computazionale è $O(n^2)$ come richiesto?

Esercizio svolto 10 (4)

La funzione `Inserisci` richiede tempo O(m) dove m ≤ n è la lunghezza di q.
Infatti la funzione `Inserisci` scorre la lista ordinata q fino a che non trova x (e in quel caso termina) o trova il posto dove inserire x (e in quel caso lo inserisce).

La funzione principale descritta richiede Θ(n) iterazioni per scorrere p e ad ogni iterazione c'è al più un'invocazione della funzione Inserisci.

Quindi il costo totale rientra in $O(n^2)$.

Esercizio svolto 10 (5)

```
def es(p):
    q = None
    while p != None:
        if p.key%2 == 0:
            q = inserisci(q, p.key)
        p=p.next
    return q
```

```
def inserisci(q,x):
    if q == None or q.key > x:
        return Nodo(x,q)
    if q.key != x:
        q.next = inserisci(q.next, x)
    return q
```