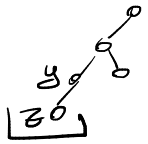
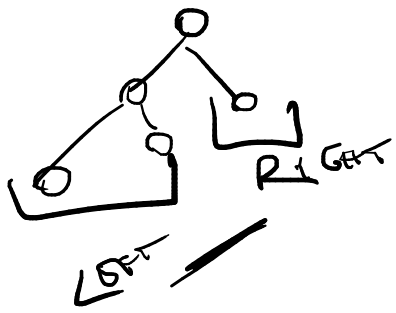
**Esercizio 1** (9 punti) Realizzare un arricchimento degli alberi binari di ricerca che permetta di ottenere per ogni nodo $x$, in *tempo costante*, il numero delle foglie nel sottoalbero radicato in $x$.

Indicare quali campi occorre aggiungere ai nodi. Fornire il codice per la funzione `leaves(x)` che restituisce il numero delle foglie nel sottoalbero radicato in $x$ e per la procedura di inserimento di un nodo `insert(T,z)`. Per entrambe valutare la complessità.



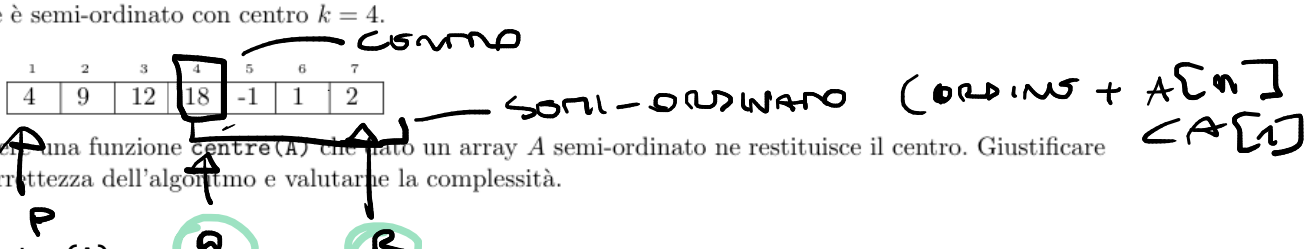
```
insert(T, z)

x = T.root
y = nil
z.leaves = 1
while x ≠ nil
      y = x
      // BST da mantenere
      if z.key < x.key
            x = x.left
      else
            x = x.right
// aggiornare z
z.p = y

if y = NIL
      T.root = z
else if z.key < y.key
      // BST da mantenere
      y.left = z
else
      y.right = z
```

```
UPDATE-LEAVES(T, z)

x = z.p

while x ≠ nil
     left_leaves = LEAVES(x.left)
     right_leaves = LEAVES(x.right)
     x.leaves = left_leaves +
                right_leaves
     x = x.p
```



```
UPDATE-LEAVES-PATH(T, z)
1.  x = z.p
2.  while x ≠ NIL
3.      old_leaves = x.leaves
4.      left_leaves = 0
5.      right_leaves = 0
6.      if x.left ≠ NIL
7.          left_leaves = x.left.leaves
8.      if x.right ≠ NIL
9.          right_leaves = x.right.leaves
10.     if x.left = NIL and x.right = NIL
11.         x.leaves = 1    // x è foglia
12.     else
13.         x.leaves = left_leaves + right_leaves
14.     x = x.p
```

**Esercizio 1** (10 punti) Diciamo che un array senza ripetizioni $A[1,n]$ è *semi-ordinato* se esiste un indice $k$, con $1 \leq k < n$, tale che $A[k+1..n]A[1..k]$ sia ordinato, ovvero i sottoarray $A[k+1..n]$ e $A[1..k]$ sono ordinati e $A[n] < A[1]$. In questo caso l'indice $k$ viene detto il centro dell'array. Ad esempio l'array che segue è semi-ordinato con centro $k = 4$.



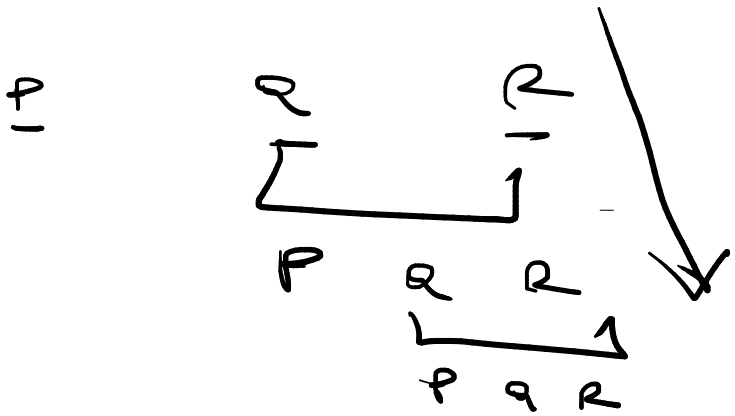

Scrivere una funzione `centre(A)` che dato un array $A$ semi-ordinato ne restituisce il centro. Giustificare la correttezza dell'algoritmo e valutarne la complessità.

```
centre(A)
     return centre-rec(A, 1, A.length)

centre-rec(A, p, r)
     // caso base

     q = (p+r)//2
     if (A[q] > A[r])
         return centre-rec (A, q+1, r)
     else
         return centre-rec (A, q, p-1)
```

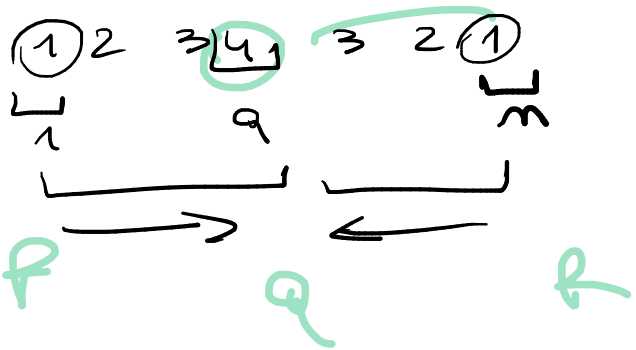
**Soluzione:** L'idea è quella di procedere con un algoritmo divide et impera. A tal fine osservazione fondamentale è la seguente: dato un sottoarry $A[p..r]$ semi-ordinato, se lo divido in due sottoarray $A[p..q]$ e $A[p..r]$, allora uno solo dei due è semi-ordinato (quello che contiene il centro), mentre l'altro è ordinato. Quando la dimensione del sottoarray $A[p..r]$ diventa 2, il centro è $p$.

Lo pseudo-codice è quindi il seguente:

```
centre(A)
    return centre-rec(A,1,A.length)

centre-rec(A,p,r)
    if r == p+1
        return p
    else
        q = (p+r)//2
        if (A[q]<A[p])
            return centre-rec(A,p,q)
        else
            return centre-rec(A,q,r)
```



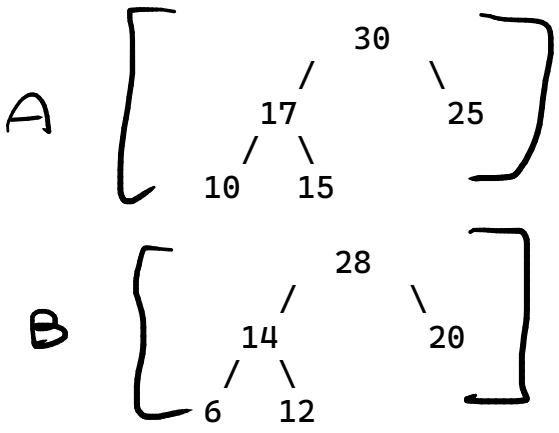

**Esercizio 1** (10 punti) Un array di interi `A[1..n]` si dice *triangolare* se esiste $q \in [1,n]$ tale che `A[1..q]` è ordinato in modo crescente e `A[q..n]` è ordinato in modo decrescente. Realizzare una funzione `maxTr(A)` che dato un array triangolare `A[1..n]`, senza elementi ripetuti, ne trova il massimo. Valutarne la complessità.




```
maxTr(A)
     return maxTr_rec(A, 1, A.length)

maxTr_rec(A, p, r)
      if p == r
         return A[p]

      q = (p + r) // 2

      if (A[q] > A[r])
          return maxTr_rec(A, q+1, r)
      else
          return maxTr_rec(A, p, q-1)
```

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

**Esercizio 1** (10 punti) Siano dati due array `A[1..2n]` e `B[1..n]` organizzati a max-heap, entrambi contenenti n elementi (`heapsize=n`). Realizzare una procedura `SortJoin(A,B,n)` che dati in input array `A` e `B` con le proprietà sopra descritte, ritorna in `A` un array ordinato contenente tutti i 2n elementi originariamente presenti in `A` e `B`. L'array `B` può essere modificato durante l'esecuzione della procedura, se necessario, ma l'algoritmo dovrà operare in *spazio costante*. Dare lo pseudocodice della procedura, motivarne la correttezza e valutarne la complessità. Se si utilizzano operazioni sui max-heap andranno definite esplicitamente.




UNICO MAX-HEAP

[30, 17, 25, 10, 15, 28, 14, 20, 6, 12]

A B

[6, 10, 12, 14, 15, 17, 20, 25, 28, 30]

FINALE (SORTED)

SPAZIO COSTANTE ( IF / ASSEGNAZIONI
( CICLI WHILE (NO FOR)

**Esercizio 1** (10 punti) Siano dati due array `A[1..2n]` e `B[1..n]` organizzati a max-heap, entrambi contenenti n elementi (`heapsize=n`). Realizzare una procedura `SortJoin(A,B,n)` che dati in input array `A` e `B` con le proprietà sopra descritte, ritorna in `A` un array ordinato contenente tutti i 2n elementi originariamente presenti in `A` e `B`. L'array `B` può essere modificato durante l'esecuzione della procedura, se necessario, ma l'algoritmo dovrà operare in *spazio costante*. Dare lo pseudocodice della procedura, motivarne la correttezza e valutarne la complessità. Se si utilizzano operazioni sui max-heap andranno definite esplicitamente.

```
SortJoin(A, B, n)

i = 1, j = 1, k = 1

while (i ≤ 2n and j ≤ n)
    if (MAX(A) ≥ MAX(B))
        A[k] = EXTRACT-MAX(A)
        i = i + 1
    else
        A[k] = EXTRACT-MAX(B)
        j = j + 1
    k = k + 1

while i ≤ 2n
        A[k] = HEAP-EXTRACT-MAX(A)
        i = i + 1
        k = k + 1

while j ≤ n
        A[k] = HEAP-EXTRACT-MAX(B)
        j = j + 1
        k = k + 1
```

MAX-HEAPIFY$(A, i)$

```
1  l = LEFT(i)
2  r = RIGHT(i)
3  if (l ≤ A.heapsize) and (A[l] > A[i])
4      max = l
5  else
6      max = i
7  if (r ≤ A.heapsize) and (A[r] > A[max])
8      max = r
9  if (max ≠ i)
10     A[i] ↔ A[max]
11     MAX-HEAPIFY(A, max)
```

BUILD-MAX-HEAP$(A)$

```
1  A.heapsize = A.length
2  for i = ⌊A.length/2⌋ down to 1
3      MAX-HEAPIFY(A, i)
```

[30, 17, 25, 10, 15, 28, 14, 20, 6, 12]

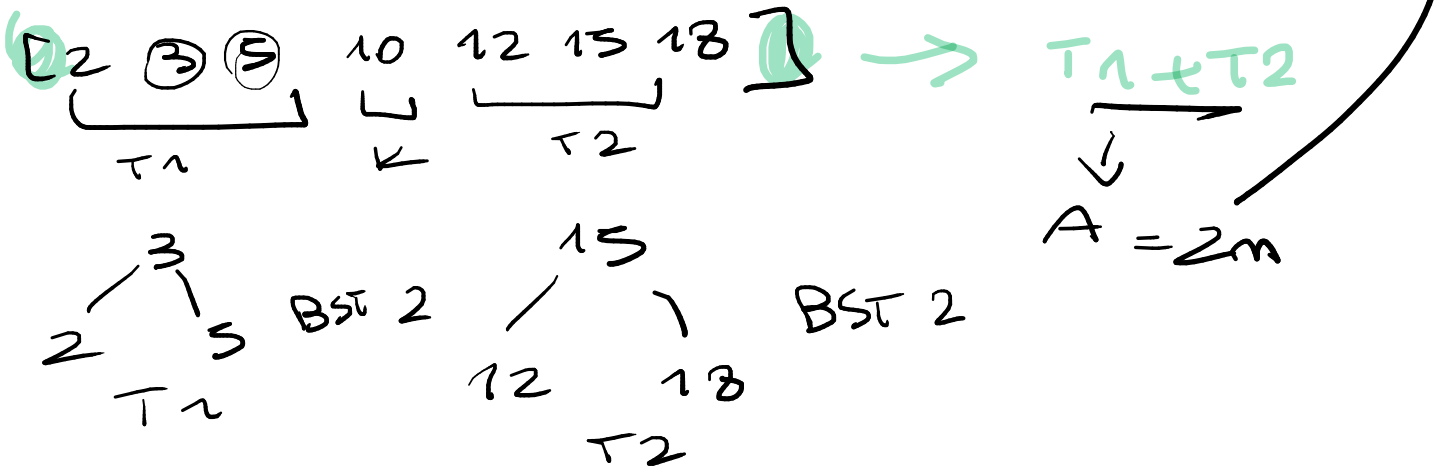
## Esercizi

**Esercizio 1** (9 punti) In questo esercizio si considerano alberi binari di ricerca (BST) completi, memorizzati in array con la stessa convenzione che si usa normalmente per la memorizzazione di uno heap in un array.

Realizzare una funzione `max(T,n)` che determina il massimo di un array `T` di dimensione `n` che memorizza un BST completo.

Scrivere quindi una funzione `merge(T1,T2,k)` che dati due array `T1` e `T2` di dimensione `n`, che memorizzano BST completi e una chiave `k` maggiore di tutte le chiavi di `T1` e minore di tutte quelle di `T2`, restituisce un array (di dimensione `2n+1`) che memorizza un BST che contiene tutte le chiavi di `T1` e `T2`, e la chiave `k`.

In entrambi i casi motivare la correttezza delle funzioni e valutarne la complessità.




MAX → ARRAY → ALBERO

[T.root / T.left / T.right]

REC ... !

```
MAX(T, n)
        x = T.root
        if x == NIL
                return NIL
        else
                while x.right ≠ nil
                        x = x.right
        return x
```

ITERATIVO ...

```
MAX(T, n)
1.  if T.root = NIL
2.      return un array vuoto
3.  A = nuovo array di dimensione n
4.  MAX-RECURSIVE(T.root, A, 0)
5.  return A
```

```
MAX-RECURSIVE(x, A, index)
1.  if x = NIL
2.      return index
3.  index = MAX-RECURSIVE(x.left, A, index)
4.  A[index] = x.key
5.  index = index + 1
6.  index = MAX-RECURSIVE(x.right, A, index)
7.  return index
```

```
MAX(T, n)
        x = T.root
        if x == NIL
                return NIL
        else
                while x.right ≠ nil
                        x = x.right
        return x
```

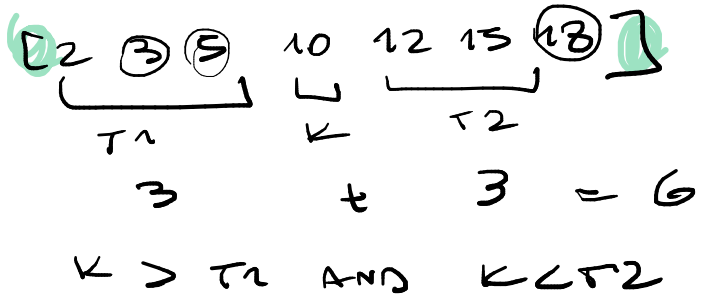
## Esercizi

**Esercizio 1** (9 punti) In questo esercizio si considerano alberi binari di ricerca (BST) completi, memorizzati in array con la stessa convenzione che si usa normalmente per la memorizzazione di uno heap in un array.

Realizzare una funzione `max(T,n)` che determina il massimo di un array `T` di dimensione `n` che memorizza un BST completo.

Scrivere quindi una funzione `merge(T1,T2,k)` che dati due array `T1` e `T2` di dimensione `n`, che memorizzano BST completi e una chiave `k` maggiore di tutte le chiavi di `T1` e minore di tutte quelle di `T2`, restituisce un array (di dimensione `2n+1`) che memorizza un BST che contiene tutte le chiavi di `T1` e `T2`, e la chiave `k`.

In entrambi i casi motivare la correttezza delle funzioni e valutarne la complessità.




```
MERGE(T1, T2, k)

i = 1, j = 1, idx = 1

A1 = MAX(T1, n)
A2 = MAX(T2, n)

while i ≤ n and j ≤ n
        if(A1[i] < A2[j])
                A[idx] = A1[i]
                i = i + 1
        else if (T1[i] >  T2[j])
                A[idx] = A2[j]
                j = j + 1
        else // caso uguale → aggiungiamo solo il più piccolo → A1
                A[idx] = A1[i]
                i = i + 1
                j = j + 1

        idx = idx + 1

// copia elementi rimanenti
while i ≤ n
        A[idx] = A1[i]
        i = i + 1
        idx = idx + 1
while j ≤ n
        A[idx] = A2[j]
        j = j + 1
        idx = idx + 1

// aggiunta di k
A[idx] = k
idx = idx + 1
INSERTION_SORT(A, k-1)
return A
```

**Esercizio 2** (10 punti) Si consideri il problema di selezione di attività compatibili, con $n$ attività $a_1, \ldots, a_n$ che ci vengono date attraverso due vettori **s** e **f** di tempi di inizio e fine, e ordinate per tempo di *inizio* (cioè $0 < s_1 \leq s_2 \leq \cdots \leq s_n$).

(a) Scrivere un algoritmo greedy iterativo che implementa la scelta greedy di selezionare l'attività che inizia per ultima.

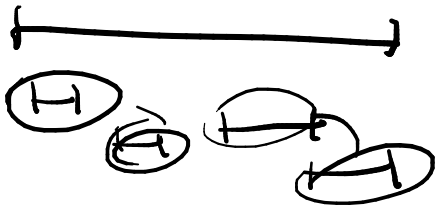
(b) Determinare l'insieme di attività restituito dall'algoritmo al punto (a) quando eseguito sul seguente insieme di 6 attività, caratterizzate dai seguenti vettori **s** e **f** di tempi di inizio e fine:

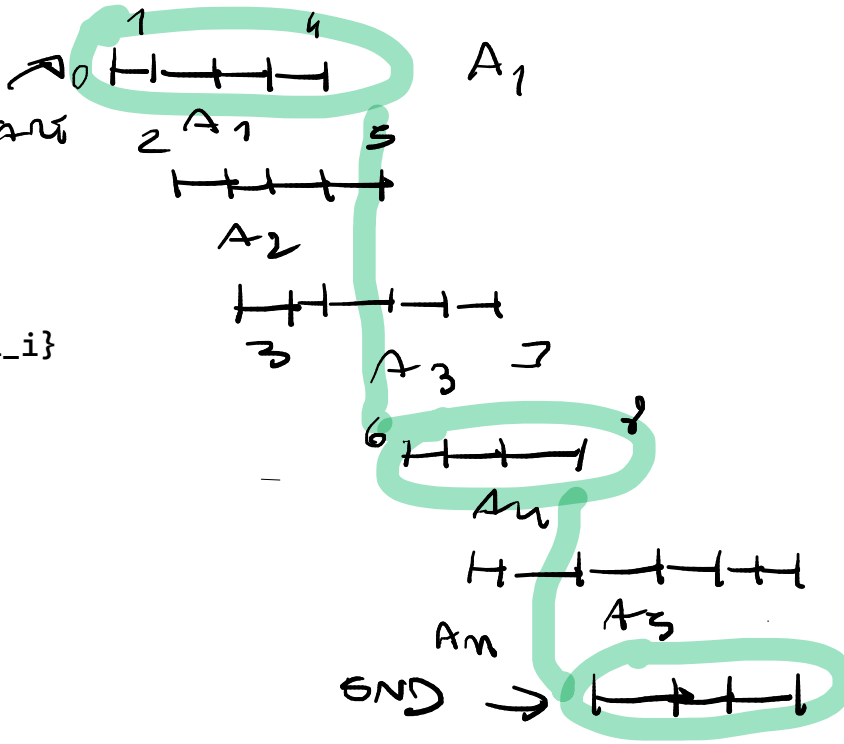
$$\mathbf{s} = (1, 2, 3, 6, 7, 9) \qquad \mathbf{f} = (4, 5, 7, 8, 12, 11)$$

(c) Dimostrare la proprietà di scelta greedy, cioè che esiste soluzione ottima che contiene l'attività che inizia per ultima.

GREEDY_SOL (S, F)

```
n = length(S)
A = {a_1}
last = 1
        for i = 2 to n
                if s_i ≥ f_last
                        A = A U {a_i}
                        last = i
return A
```






$\forall\ s[3] \leq s[m]$

$O' = O \setminus \{a_3\}$