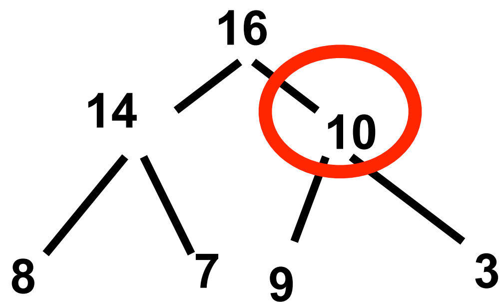
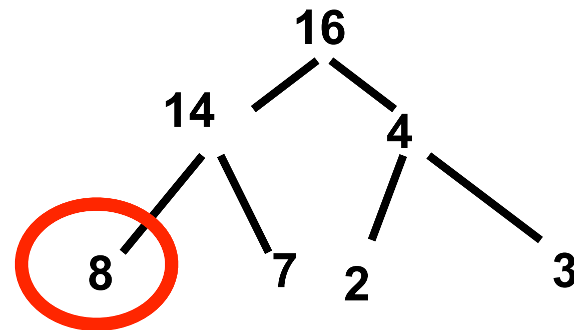
Esercizio 1.
Dato in input un max-heap di n interi non necessariamente distinti, si scriva un algoritmo che dia in output il quarto intero più grande. L'algoritmo dovrebbe lavorare in tempo costante.
Di tale algoritmo: si dia la spiegazione a parole, che contenga anche una spiegazione informale della sua correttezza, si calcoli il tempo di esecuzione asintotico (costo computazionale) e se ne fornisca una versione in pseudocodice.

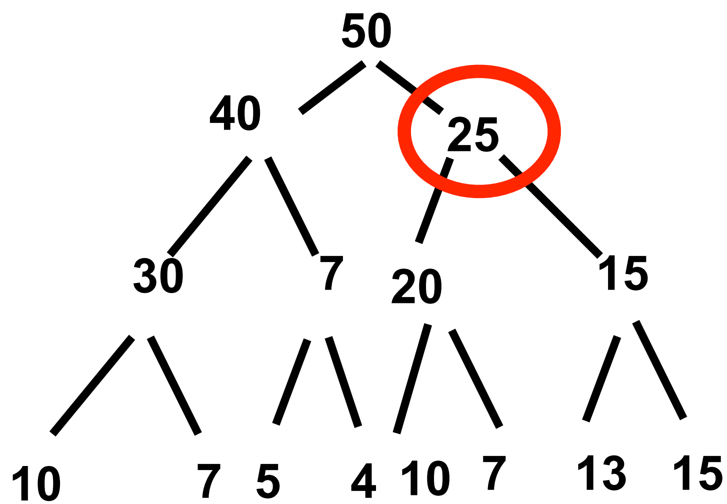
Sol. Si tratta di considerare vari casi. Per semplicità supponiamo che l'array contenga almeno $2^4 - 1$ elementi. Il quarto più grande può trovarsi nei livelli 1, 2 o 3 e non può trovarsi al livello 4 perchè un nodo in quel livello ha quattro antenati e quindi 4 elementi più grandi o uguali.
Il terzo più grande può trovarsi nel livello 1 o nel livello 2, ma non nel livello 3 perchè avrebbe tre antenati più grandi o uguali. Se il terzo più grande si trova nel livello 1, allora il massimo tra i valori del livello 2 e il fratello del terzo più grande è il quarto più grande.
Se invece il terzo più grande si trova nel livello 2, allora il quarto più grande può essere nello stesso livello o in quello inferiore.
Poiché si lavora su al più $2^4 - 1$ elementi, l'algoritmo ha tempo costante.
Per semplicità diamo prima una funzione che calcola il terzo più grande.



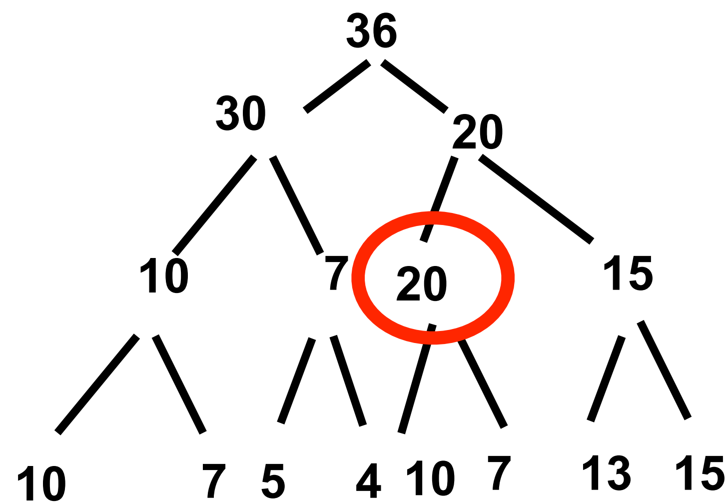
Esempio in cui il terzo più grande è nel livello 1



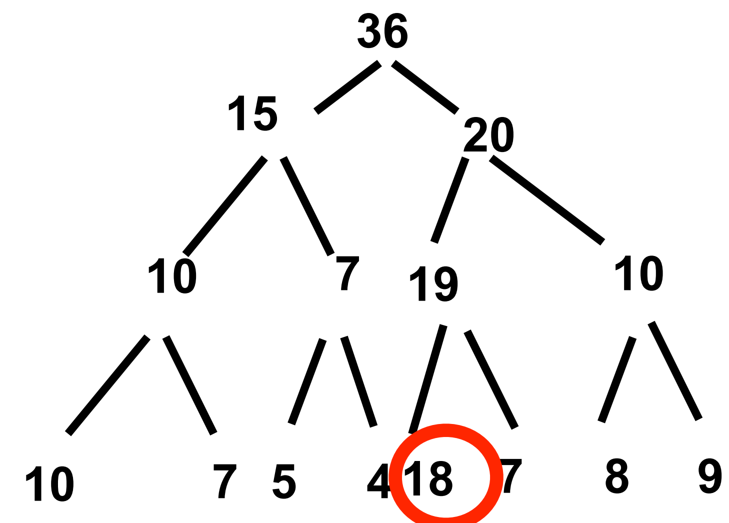
Esempio in cui il terzo più grande è nel livello 2



Esempio in cui il quarto più grande è nel livello 1



Esempio in cui il quarto più grande è nel livello 2



Esempio in cui il quarto più grande è nel livello 3

```
3PGMax-heap1(A)
input: un array di interi A
prec: gli elementi di A formano un max-heap, di almeno 2^3-1= 7 elementi
output: l'indice del terzo più grande.
j = max(A[4],A[5],A[6],A[7])
if A[2] ≥ A[3] then
            j = max(A[4],A[5])   A[6] e A[7] sono minori di A[3]
           if A[3] ≥ A[j] then  return 3 else return j
           else
           j = max(A[6],A[7]). A[4] e A[5] sono minori di A[2]
          if A[2] ≥ A[j] then  return 2 else return j
```

```
4PGMax-heap1(A)
input: un array di interi A
prec: gli elementi di A formano un max-heap, di almeno 2^4-1=15  elementi
output: l'indice del quarto più grande.
i = 3PGMax-heap(A)
if i ≤3 1 then //il terzo più grande si trova nel livello 1, allora il quarto più grande è nel livello 2
            return   max(A[4],A[5],A[6],A[7])
       else //il terzo più grande si trova nel livello 2, allora il quarto più grande può essere nel livello 1,2 o
3
            return max{A[j] | j ≠ i and j≠i/2 and 2≤j≤15}
```

Esercizio 2.
Si consideri la seguente funzione e se ne calcoli il tempo di esecuzione asintontico, giustificando a parole i passi fondamentali del calcolo.

```
void function(n) {
count = 0;
for (i=n/2; i<=n; i++)                          Θ(nlg² n)
    for (j=1; j<=n; j = 2 * j)                   Θ(lg² n)
        for (k=1; k<=n; k = k * 2)               Θ(lg n)
            count++;
}
```

Il ciclo for più interno è eseguito lg n volte circa, perché il parametro k parte da 1 e deve diventare la prima potenza di 2 maggiore o uguale a n per provocare l'uscita dal ciclo. Osservando che ogni intero è compreso tra due potenze di 2 consecutive si ha $2^h \leq n < 2^{h+1}$, è evidente che il ciclo viene eseguito h volte. Per la stessa ragione il ciclo immediatamente più esterno viene eseguito lg n volte e quindi i due annidati hanno un tempo di esecuzione pari a $\Theta(\lg^2 n)$. Infine il ciclo più esterno viene ripetuto n/2 volte e quindi il tutto ha tempo di esecuzione $\Theta(n\lg^2 n)$.

Esercizio 3.
Si considerino due array A e B ordinati di n e m numeri interi positivi, rispettivamente.
Si scriva un algoritmo che dia in output la coppia di indici che individuano un elemento di A e un elemento di B la cui differenza in valore assoluto è 3, se presente, e la coppia (-1,-1) altrimenti.
Di tale algoritmo: si dia la spiegazione a parole, che contenga anche una spiegazione informale della sua correttezza, si calcoli il tempo di esecuzione asintotico (costo computazionale) e se ne fornisca una versione in pseudocodice.

Sol 1. Questa soluzione è un'applicazione della ricerca binaria. Per ogni elemento dell'array con meno elementi, supponiamo sia B, si cerca in A un elemento a distanza 3 e se lo si trova si dà in output la coppia di indici relativi ai due elementi.
Detto n il numero degli elementi di A e m quello di B, sotto l'ipotesi che $n > m$ il tempo di esecuzione nel caso peggiore è dunque $\Theta(m \lg n)$ e quindi in $O(m \lg n)$ in tutti i casi.

Useremo la versione seguente della ricerca binaria:

BinS(A,k)

INPUT: una sequenza A di elementi e l'elemento da cercare, k.

PREC: A[0]≤A[1]≤...≤A[*n*-1], dove *n* è il numero degli elementi

OUTPUT: se k è presente, restituisce la posizione di k in A, altrimenti -1

```
lo = 0 e hi = len(A)
%nel ciclo che segue l'intervallo determinato dagli indici lo,…,hi-1 è quello in
   cui trovare l'elemento, se presente
while  lo < hi:
     m = lo + (hi-lo)/2
     if k = A[m] return m
     if k < A[m] then hi=m
%cerca k tra gli elementi di indice lo,…,m-1,
     if k > A[m]   then lo=m+1
%cerca k tra gli elementi di indice m+1,…,hi-1
return -1
```

Dist3(A,B)

INPUT: due array di interi positivi A e B.

PREC: A e B sono ordinati crescenti e A ha un numero di elementi maggiore o uguale a B

OUTPUT: la coppia di indici che individuano un elemento di A e un elemento di B la cui differenza in valore assoluto è 3, se presente, e la coppia (-1,-1) altrimenti.

```
n = A.length
m = B.length
for  j  = 0 to m do
      i = BinS(A,B[j]+3)
     if i ≠ -1 then return (i,j)
     if B[j] - 3 > 0 then i = BinS(A,B[j]-3)
     if i ≠ -1 then return (i,j)
return (-1,-1)
```

Sol 2.

Si tratta di un’applicazione della funzione fondi. Prima diamo la soluzione nel caso gli elementi siano tutti distinti poi faremo vedere come estendere la soluzione al caso di array con ripetizioni. Si scorrono i due array A e B e quando si considera $A[i] - B[j] = d$ si possono dare vari casi:

1. se $d = 3$ oppure $d = -3$ si danno in output i due indici,
2. $d > 3$, si deve incrementare j per confrontare $B[j+1]$ con $A[i]$ visto che $B[j+1] > B[j]$ e quindi potrebbe trovarsi a distanza 3, mentre $A[i+1]$ è maggiore di $A[i]$ e la sua differenza con $B[j]$ non può che essere maggiore di d e quindi di 3.
3. $0<d<3$, allora vuol dire che $A[i] = B[j] + d$ e quindi $B[j+1]$ potrebbe essere a distanza 3 da $A[i]$, così come $B[j+2]$ e $B[j+3]$, quindi prima di incrementare i bisogna controllare quei valori e in caso di uguaglianza, per esempio se $A[i] = B[j+1]$ si devono fare i controlli del caso 6.
4. $d < -3$, è analogo al caso 2 scambiando ruoli di A e B, infatti $A[i] = B[j] - d$,
5. $-3<d<0$ è analogo al caso 3, scambiando i ruoli di A e B,
6. Se $d=0$ potrebbe essere corretto incrementare i, ma anche j, quindi dobbiamo confrontare $A[i]$ e $B[j]$ con i successivi 3 in entrambi gli array. Basta farlo con tre perché $A[i+1] >A[i]$ e quindi $A[i+1] \geq A[i] +1$ e $A[i+2] \geq A[i+1] +1 \geq A[i] +2$, e ancora $A[i+3] \geq A[i+2]+1 \geq A[i]+3$, quindi $A[i+4] > A[i]+3$. Lo stesso vale per $B[j]$. Una volta verificato che non c’è un elemento a distanza 3 con $A[i]=B[j]$ allora incrementiamo sia i che j e consideriamo il confronto successivo.

Vediamo ora come trattare i duplicati, in modo da non aumentare la complessità e non perdere soluzioni.
Ogni volta che si considera una coppia A[i] e B[j] si controlla se ci sono elementi uguali ad A[i] alla sua destra e uguali a B[j], alla sua destra facendo in modo che il valore finale dei due indici sia quello dell'ultimo elemento a destra uguale ad A[i] o a B[j], ora sappiamo che il successivo dell'elemento individuato è diverso da A[i] così come il successivo lo è da B[j], quindi siamo ricondotti al caso precedente.
Useremo la seguente funzione per trattare i duplicati

```
ElDuplicati(C,i)
Input: un array di interi e un intero i
Prec: C è ordinato e 0≤i≤C.length
Output: l'indice dell'ultimo elemento di C alla destra di C[i] uguale ad C[i],
in altre parole un indice j tale che j ≥ i  e C[k] = C[i] per k = i,…,j
k=i
        while  (k < n-1 and C[k] = C[k+1]) then k++
controllo le copie di C[i]
return k
```

Questa soluzione è in Θ(n+m), perchè al più i due array sono uguali e di elementi diversi, quindi ogni volta la differenza è 0 e bisogna esaminare al più i tre successivi. Ogni elemento duplicato di un array viene confrontato con tutte le sue copie ma poi solo la copia più a destra è confrontata con un elemento dell'altro array.

Useremo anche la seguente funzione per esaminare i tre elementi successivi a uno considerato

Succ3(C,D,i,j)
Input: due array di interi e due interi i e j
Prec: C e D sono ordinati, 0≤i≤C.length 0≤j≤D.length
Output: verifica i tre elementi di D alla destra di D[j] confrontandoli con C[i], se una di queste coppie è a distanza tre dà in output la coppia di indici relativa, altrimenti la coppia (-3,j), mentre la coppia (-2,h+k) è data in output quando uno degli elementi a destra di D[j] ed esattamente il k-simo, risulta uguale a C[i], perchè in tal caso è necessario anche controllare, prima di incrementare i, che non ci sia in C oltre che in D un elemento a distanza 3 da C[i].

```
        k=1
        h=j
        while (k + h < n-1 and k ≤ 3 and C[i] - D[k+h] ≤ 3 then verifica B[j+k] - A[i], per
  k=1,2,3.
                h = ElDuplicati(D,k+h) - k
                if (C[i] - D[k+h] = 3 or C[i] - D[k+h]  = -3 ) then return (i,h+k)
                else
                        if C[i] - D[k+h] = 0 then return (-2,h+k)
               bisogna andare al caso d=0 con l'indice k+h per C e i invariato
                k++
            return (-3,j)  nel caso si esca dal while senza aver trovato coppie a distanza 3 o
a distanza 0
```

Useremo infine la seguente funzione per esaminare i tre elementi successivi a due uguali nei due array

Dist0(C,D,i,j)
Input: due array di interi e due interi i e j
Prec: C e D sono ordinati, 0≤i≤C.length, 0≤j≤D.length e C[i] = D[j]
Output: verifica i tre elementi di C alla destra di C[i] confrontandoli con C[i], se una di queste coppie è a distanza tre dà in output la coppia di indici relativa, e fa lo stesso per i tre elementi alla destra di D[j], altrimenti dà in output (-2,j)

```
k=1
h=i
while (k + h < n-1 and k ≤ 3 and A[k+h] - A[i]) ≤ 3 then   verifica A[i+k] - A[i], per k=1,2,3.
    h = ElDuplicati(A,k+h) - k
    if (A[k+h] - A[i]) = 3 or A[k+h] - A[i]) = -3 ) then return (k+h,j)
    k++
k=1
h=j
while (k + h < n-1 and k ≤ 3 and B[k+h] - B[h]) ≤ 3 then verifica B[i+k] - A[i], per k=1,2,3.
    h = ElDuplicati(B,k+h) - k
    if (B[k+h] - B[j] = 3 or B[k+h] - B[j] = -3 ) then return (i,h+k)
    k++
return (-2,j)
```

Elementi a distanza 3: pseudocodice

```
Dist3(A,B)
input: due array di interi
prec: gli elementi di A e di B sono positivi e ordinati in ordine crescente
output: dà in output  la coppia di indici che individuano una coppia (A[i],B[j]) la cui differenza  in valore assoluto è 3, (-1,-1) altrimenti.
/è una versione modificata della funzione fondi, utilizzata per fondere due array ordinati in un unico array ordinato.
i=0, j=0
n = A.length
m = B.length
while (i < n and j < m) do
        i = ElDuplicati(A,i)
        j = ElDuplicati(B,j)
        d = A[i] - B[j]
        if d = 3 or d = -3 ) then return (i,j)
caso 1 if ( d > 3 ) then j++  in tal caso A[i] > B[j] + 3  quindi A[i+1] è ancora più distante
            else
                if ( d > 0 ) then
                              (s,t) = Succ3(A,B,i,j)  in tal caso A[i] > B[j] + d  quindi B[j+1], B[j+2] e B[j+3]  vanno confrontati con A[i]
prima di incrementare i
                              if (s≥0) then return (s,t)
                              if (s=-3) then i++
                              if (s=-2) then (s,t) = Dist0(A,B,i,t)
                              if (s≥0) then return (s,t)
                              if (s=-2) then  i++
caso 2 if  (d < - 3) then i++  in tal caso B[j] > A[i] +3  quindi B[j+1] è ancora più distante
            else
                if ( d < 0 ) then
                              (s,t) = Succ3(B,A,j,i) in tal caso B[j] > A[i] + d  quindi A[i+1], A[i+2] e A[i+3]  vanno confrontati con B[j],
prima di incrementare j
                              if (s≥0) then return (s,t)
                              if (s=-3) then j++
                              if (s=-2) then (s,t) = Dist0(A,B,t,j)
                              if (s≥0) then return (s,t)
                              if (s=-2) then  j++
caso 3 if ( d = 0 ) then    B[j] = A[i]
            (s,t) = Dist0(A,B,i,j)
             if (s≥0) then return (s,t)
        i++
        j++
while  i < m  do   uscita per j=m, si confrontano i rimanenti in A con l'ultimo di  B,
          If (A[i] - B[n-1]) =3 then return (i,n-1);
           i=i+1;
 while  j < n  do   uscita per i=m, si confrontano i rimanenti in B con l'ultimo di  A,
          If (A[m-1] - B[j]) then return (m-1,j);
          j=j+1;
```