1 Soluzioni

1.1 Differenze minime e massime

min-gap : Il problema ha ovviamente una soluzione di complessità $O(n^2)$, che confronta tutte le possibile coppie di indici e i rispettivi valori. E' possibile fare meglio?
E' possibile notare che ordinando i valori, è possibile scorrere il vettore confrontando posizioni consecutive, alla ricerca della coppia di valori più vicina. Così facendo però si perde l'informazione sulla posizione dei valori nel vettore originale. In ogni caso, l'ordinamento costa $O(n \log n)$; una volta individuati i due valori più vicini, è possibile cercarli in tempo $O(n)$ nel vettore originale.

```
(integer, integer) min-gap(integer[] A, integer n)
    integer[] B ← newinteger[1...n]
    for i ← 1 to n do B[i] ← A[i]
    quicksort(B, n)

    integer min ← 2
    for i ← 3 to n do
        if N[i] − B[i − 1] < B[min] − B[min − 1] then
            min ← i

    integer i1 ← 0
    integer i2 ← 0
    for i ← 1 to n do
        if i1 = 0 and A[i] = A[min − 1] then
            i1 ← i
        else if i2 = 0 and A[i] = A[min] then
            i2 ← i
    return (i1, i2)
```

Altrimenti, si potrebbe modificare un algoritmo di ordinamento in modo da mantenere, parallelamente ai valori, la loro posizione all'interno del vettore originale. Una volta ordinati i valori, è sufficiente scorrerli tutti cercando la coppia di elementi consecutivi con differenza minima.
La complessità dell'algoritmo risultante è dominata dall'ordinamento, che ha costo $O(n \log n)$.

max-gap : Il problema è più semplice; si tratta di individuare l'indice in cui si trova l'elemento minimo e l'indice in cui si trova l'elemento massimo.

```
(integer, integer) max-gap(integer[] A, integer n)
    integer i1 ← min(V, n);      % Restituisce indice del minimo
    integer i2 ← max(V, n);      % Restituisce indice del massimo
    return (i1, i2)
```

Ovviamente un algoritmo del genere ha costo lineare nel numero di elementi $O(n)$.

1.2 Per fare un albero (binario di ricerca) ci vuole...

L'idea è prendere il valore nella posizione mediana e inserirlo nell'albero; questo fa sì che i due sottoalberi sinistro e destro abbiano dimensioni simili (differiscono al più di 1) e quindi se costruiti in maniera bilanciata, fanno sì che l'albero risultante sia bilanciato.

(**integer**, **integer**) build-tree-rec(TREE T, **integer**[ ] A, **integer** i, **integer** j)

```
if i < j then
    integer m ← (i + j)/2
    TREE u ← T.insertNode(V[m], V[m])
    build-tree-rec(u, A, i, m − 1)
    build-tree-rec(u, A, m + 1, j)
```

(**integer**, **integer**) build-tree-rec(**integer**[ ] A, **integer** n)

```
TREE T ← Tree()
build-tree-rec(T, A, 1, n)
```

Alcune cose da notare:

- Nel libro, insertNode() ritorna il puntatore alla radice dell'albero. In questo caso, il costo computazionale dell'algoritmo è $O(n \log n)$, perchè ogni inserimento deve trovare la posizione corretta a partire dalla radice (costo $O(\log n)$)
- Se invece si assume che insertNode() ritorni il nuovo nodo appena creato, la complessità dell'algoritmo è $O(n)$, perchè ogni inserimento deve semplicemente scegliere se mettere un valore a destra o a sinistra. Se così è, bisogna specificarlo nel compito.
- Una possibilità alternativa è scrivere codice ad-hoc per la gestione dell'albero per questo problema. La mia versione tende a minimizzare il codice scritto.
- Notate che rispetto alla versione che ho presentato a lezione, ho aggiunto l'ipotesi che i numeri siano distinti. Questo semplifica la gestione degli ABR.

1.3 Ricorrenza

Bisogna affrontare i due casi (pari e dispari) separatamente; utilizzando i teoremi visti a lezione, è facile vedere che $T(n) = \Theta(\log n)$ nel caso delle potenze di 2, mentre $T(n) = \Theta(n)$ nel caso dei numeri dispari (infatti, nel caso delle potenze di 2 si applica solo la divisione per 2, che produce ancora una potenza di 2, mentre nel caso dei numeri dispari si applica solo la sottrazione di 2, che produce ancora un numero dispari). Cercando un limite inferiore e superiore, è facile quindi capire che un limite inferiore per la ricorrenza è pari a $\Omega(\log n)$, mentre un limite superiore è $O(n)$.
Dimostriamo per sostituzione il limiter superiore: vogliamo dimostrare che $\exists c > 0, \exists m \geq 0$ tale che $T(n) \leq cn$ per ogni $n \geq m$.
Se n è pari,

$$\begin{aligned} T(n) &= T(n/2) + 1 \\ &\leq cn/2 + 1 \\ &\leq cn \end{aligned}$$

che è vera per $c \geq 2/n$, che è soddisfacibile per $c = 1$ con $m = 2$.

Se n è dispari,

$$\begin{aligned} T(n) &= T(n-2)+1 \\ &\leq c(n-2)+1 \\ &\leq cn \end{aligned}$$

che è vera per $2c \geq 1$, ovvero per $c \geq 1/2$ per tutti gli n.
Il caso base è vero per $T(n) = 1 \geq 1 \cdot c$, ovvero per $c \geq 1$. Quindi è banalmente dimostrato che per $c = 1$, $m = 2, T(n) \leq cn$ per ogni $n \geq m$.
La dimostrazione per il limite inferiore è analoga: dobbiamo dimostrare che $\exists c > 0, \exists m \geq 0$ tale che $T(n) \geq c \log n$ per ogni $n \geq m$.
Se n è pari,

$$\begin{aligned} T(n) &= T(n/2)+1 \\ &\geq c\log(n/2)+1 \qquad \text{per sostituzione} \\ &= c\log n - c + 1 \\ &\geq c\log n \end{aligned}$$

che è vera per tutti i $c \leq 1$ e per tutti gli n.
Se n è dispari,

$$\begin{aligned} T(n) &= T(n-2)+1 \\ &\geq c\log(n-2)+1 \qquad \text{per sostituzione} \\ &\geq c\log(n/2)+1 \qquad \text{per } m \geq 4 \\ &\geq c\log n \end{aligned}$$

che è vera per tutti i $c \leq 1$ e per tutti gli $n \geq m = 4$.
Il caso base dà origine a problemi, ma è sufficiente utilizzare $n = 2, 3$ come casi base e vedere che

$$\begin{aligned} T(2) = T(1)+1 &= 2 \geq 2c \\ T(3) = T(1)+1 &= 2 \geq 3c \end{aligned}$$

che sono entrambe soddisfatte per $c \geq 1$.
Quindi, per $c = 1, m = 4$, il nostro limite inferiore è provato.