1 Massimo Sottovettore Ordinato

Dato un array di n numeri interi, trovare il sottovettore continuo più lungo che sia ordinato in modo strettamente crescente. Il sottovettore continuo è una sequenza di elementi consecutivi nell'array originale. L'obiettivo è progettare un algoritmo con complessità temporale lineare, ovvero $O(n)$.

Soluzione

Per risolvere questo problema in $O(n)$, possiamo utilizzare un approccio a scansione lineare che attraversa l'array una volta, tenendo traccia della lunghezza del sottovettore crescente corrente e aggiornando il sottovettore massimo trovato finora.

Ecco i passi dell'algoritmo:

1. Inizializza le variabili:
   - `max_inizio` e `max_fine` per memorizzare l'inizio e la fine del sottovettore crescente massimo trovato finora.
   - `max_lunghezza` per tenere traccia della sua lunghezza.
   - `corrente_inizio` e `corrente_lunghezza` per tenere traccia dell'inizio e della lunghezza del sottovettore crescente corrente.
2. Scorri l'array:
   - Confronta ciascun elemento con il precedente:
   - Se l'elemento corrente è maggiore del precedente, incrementa `corrente_lunghezza`.
   - Se l'elemento corrente non è maggiore, verifica se il sottovettore crescente corrente è il più lungo trovato finora. Se lo è, aggiorna `max_lunghezza`, `max_inizio` e `max_fine`. Poi, resetta `corrente_lunghezza` e `corrente_inizio` per il nuovo sottovettore crescente.
3. Dopo la scansione, verifica nuovamente se il sottovettore crescente corrente è il più lungo trovato finora.
4. Restituisci il massimo sottovettore crescente individuato.

Pseudocodice

```
ALGORITMO MaxSottovettoreCrescente(arr)
    SE lunghezza(arr) = 0 ALLORA
        RESTITUISCI []
```

```
    max_inizio ← 0
    max_fine ← 0
    max_lunghezza ← 1

    corrente_inizio ← 0
    corrente_lunghezza ← 1

    PER i DA 1 A lunghezza(arr) - 1 FAI
        SE arr[i] > arr[i-1] ALLORA
            corrente_lunghezza ← corrente_lunghezza + 1
        ALTRIMENTI
            SE corrente_lunghezza > max_lunghezza ALLORA
                max_lunghezza ← corrente_lunghezza
                max_inizio ← corrente_inizio
                max_fine ← i - 1
            FINE SE
            corrente_inizio ← i
            corrente_lunghezza ← 1
        FINE SE
    FINE PER

    SE corrente_lunghezza > max_lunghezza ALLORA
        max_inizio ← corrente_inizio
        max_fine ← lunghezza(arr) - 1

    RESTITUISCI arr[max_inizio ... max_fine]
FINE ALGORITMO
```

Complessità

L'algoritmo ha complessità temporale $O(n)$ poiché scorre l'array una sola volta per identificare il sottovettore crescente più lungo.

2 Somma a Valore Target Dato

Dato un array di n numeri interi, vogliamo determinare se esistono due elementi distinti nell'array che sommano a un valore target dato. L'obiettivo è progettare un algoritmo con complessità temporale pari a $O(n \log n)$ o migliore.

Soluzione

Per risolvere questo problema in $O(n \log n)$, possiamo usare un approccio che combina l'ordinamento e la tecnica dei due puntatori. Di seguito i passi dell'algoritmo:

1. Ordina l'array in ordine crescente usando un algoritmo di ordinamento efficiente come il Merge Sort o il Quick Sort. Questo richiede $O(n \log n)$.
2. Inizializza due puntatori:
   - Uno punta all'inizio dell'array (indice 0).
   - L'altro punta alla fine dell'array (indice $n - 1$).
3. Calcola la somma dei valori puntati dai due puntatori:
   - Se la somma è uguale al target, allora hai trovato due numeri che soddisfano il criterio.
   - Se la somma è minore del target, incrementa il puntatore sinistro (spostandolo verso destra) per aumentare la somma.
   - Se la somma è maggiore del target, decrementa il puntatore destro (spostandolo verso sinistra) per diminuire la somma.
4. Continua il processo fino a quando i puntatori si incrociano (cioè $i < j$).

Se i puntatori si incrociano senza trovare una somma che corrisponde al target, allora non esistono due elementi distinti che soddisfano il requisito.

Pseudocodice

```
ALGORITMO HasPairWithSum(arr, target)
    Ordina(arr)  // Ordina l'array in O(n log n)
    i ← 0
    j ← lunghezza(arr) - 1

    MENTRE i < j FAI
```

```
        current_sum ← arr[i] + arr[j]
        SE current_sum = target ALLORA
            RESTITUISCI (arr[i], arr[j])  // Coppia trovata
        ALTRIMENTI SE current_sum < target ALLORA
            i ← i + 1  // Incrementa il puntatore sinistro
        ALTRIMENTI
            j ← j - 1  // Decrementa il puntatore destro
        FINE SE
    FINE MENTRE

    RESTITUISCI None
FINE ALGORITMO
```

Complessità

L'algoritmo ha complessità $O(n \log n)$ per via dell'ordinamento, seguito da una ricerca lineare $O(n)$ con due puntatori. Pertanto, la complessità totale è $O(n \log n)$.

3 Unione di k array ordinati

Dati k array ordinati, ognuno di lunghezza n, scrivi un algoritmo per unirli in un unico array ordinato con complessità $O(n \log k)$. L'algoritmo deve utilizzare una coda con priorità (min-heap) per eseguire l'unione.

Soluzione

L'idea principale è utilizzare una *coda con priorità* (min-heap) per mantenere i primi elementi di ciascun array e inserire progressivamente l'elemento più piccolo nel nuovo array di output. In questo modo, otteniamo una complessità $O(n \log k)$, dove k è il numero di array e n è la lunghezza totale degli array combinati.

Un **min-heap** ci permette di:

- Mantenere sempre a portata di mano l'elemento più piccolo tra quelli presenti nei k array.
- Effettuare l'operazione di estrazione dell'elemento minimo (l'elemento più piccolo tra tutti) in tempo $O(\log k)$.

Per unire k array ordinati, dobbiamo continuamente trovare il valore più piccolo tra quelli ancora non inseriti nel risultato. Inserendo nel min-heap il primo elemento di ciascun array, possiamo estrarre il minimo in $O(\log k)$, inserire tale elemento nell'array risultato e aggiungere al min-heap il successivo elemento dell'array da cui proveniva. Grazie al min-heap, riduciamo il tempo di ricerca del minimo a $O(\log k)$ invece di $O(k)$, il che migliora l'efficienza complessiva dell'algoritmo.

1. Inizializza un min-heap.
2. Inserisci nel min-heap il primo elemento di ciascun array, mantenendo anche un riferimento all'array da cui proviene e all'indice dell'elemento.
3. Esegui un ciclo finché il min-heap non è vuoto:
    - Estrai il minimo dal min-heap (sarà l'elemento più piccolo tra quelli attualmente nei k array) e aggiungilo all'array di output.
    - Se l'elemento estratto non è l'ultimo dell'array da cui proviene, inserisci nel min-heap il successivo elemento dello stesso array.

Pseudocodice

```
ALGORITMO UnisciKArrayOrdinati(arrays)
    risultato ← array vuoto
    min_heap ← nuova coda con priorità (min-heap)

    // Inserisci il primo elemento di ciascun array nel min-heap
    PER i DA 0 A k - 1 FAI
        SE arrays[i] non è vuoto ALLORA
            AGGIUNGI (arrays[i][0], i, 0) al min_heap
        FINE SE
    FINE PER

    // Unisci gli array
    MENTRE min_heap non è vuoto FAI
        elemento, arr_indice, elem_indice ← EstraiMinimo(min_heap)
        AGGIUNGI elemento a risultato

        // Inserisci il prossimo elemento dell'array nel min-heap, se esiste
        SE elem_indice + 1 < lunghezza(arrays[arr_indice]) ALLORA
            AGGIUNGI (arrays[arr_indice][elem_indice + 1], arr_indice, elem_indice + 1)
        FINE SE
    FINE MENTRE

    RESTITUISCI risultato
FINE ALGORITMO
```

Complessità

- **Tempo**: L'algoritmo richiede $O(n \log k)$, poiché ogni inserimento o estrazione dal min-heap richiede $O(\log k)$ operazioni, e ci sono in totale n elementi da inserire.
- **Spazio**: L'algoritmo utilizza $O(k)$ spazio aggiuntivo per il min-heap, poiché tiene contemporaneamente in memoria solo un elemento di ciascun array.

4 Ordinamento... Quasi Lineare!

Dato un array di numeri interi non negativi, implementa una funzione che li ordini utilizzando **Radix Sort**. L'algoritmo deve considerare ogni cifra, ordinando prima le unità, poi le decine, centinaia, ecc., fino alla cifra più significativa.

Esempio:

- Input: $[170, 45, 75, 90, 802, 24, 2, 66]$
- Output atteso: $[2, 24, 45, 66, 75, 90, 170, 802]$

Soluzione

La soluzione usa *Radix Sort*, che ordina i numeri considerando le cifre da quella meno significativa alla più significativa, con l'aiuto di *Counting Sort* per ogni posizione.

Passaggi della soluzione: 1. Trova il valore massimo nell'array per sapere quante cifre ha il numero più lungo. 2. Applica *Counting Sort* per ciascuna cifra, partendo dalle unità fino alla cifra più significativa. 3. Ripeti *Counting Sort* su ogni posizione decimale (unità, decine, centinaia, ecc.) fino a ordinare completamente l'array.

4.1 Pseudocodice

```
def counting_sort(arr, exp):
    n = len(arr)
    output = [0] * n
    count = [0] * 10  # Conta cifre da 0 a 9

    # Conta il numero di occorrenze di ciascuna cifra
    for i in range(n):
        index = (arr[i] // exp) % 10
        count[index] += 1

    # Trasforma count per posizioni cumulative
    for i in range(1, 10):
        count[i] += count[i - 1]

    # Costruisce l'output ordinato
    i = n - 1
    while i >= 0:
        index = (arr[i] // exp) % 10
        output[count[index] - 1] = arr[i]
```

```
        count[index] -= 1
        i -= 1

    # Copia l'output nell'array principale
    for i in range(n):
        arr[i] = output[i]

def radix_sort(arr):
    max_element = max(arr)
    exp = 1  # Inizia con la cifra delle unit
    while max_element // exp > 0:
        counting_sort(arr, exp)
        exp *= 10  # Passa alla cifra successiva
```

Costo Computazionale

Complessità Temporale: Radix Sort ha complessità $O(d \cdot (n + b))$, dove n è il numero di elementi nell'array, d è il numero di cifre del numero più grande e b è la base del sistema numerico (in questo caso, 10 per le cifre decimali). Con b costante, la complessità diventa $O(n \cdot d)$, che è quasi lineare rispetto alla lunghezza dell'array.

Complessità Spaziale: Utilizza $O(n)$ spazio extra per mantenere l'array di output durante *Counting Sort* per ciascuna cifra, quindi la complessità spaziale è $O(n)$.