Strutture Dati, Algoritmi e Complessità
Esercitazione del 29 Aprile 2025

In questa esercitazione, lo scheletro del codice sarà fornito solo in JAVA. Gli studenti che intendono lavorare in C possono farlo, replicando lo schema proposto. Scopo dell'esercitazione è risolvere un problema di studio reale, investigando diverse tecniche algoritmiche e la loro complessità.

1 Descrizione del problema

Siete stati assunti privatamente da Daniel Rossi, un ex professionista del gioco d'azzardo che intende utilizzare informazioni riservate per massimizzare i suoi profitti in un casinò. Il casinò in questione, uno dei più noti di Las Vegas, è gestito da *T.B. Enterprises*. Daniel ha scoperto che alcune irregolarità nei meccanismi di gioco gli permettono di prevedere con una certa precisione i risultati delle mani al tavolo del Black Jack. In particolare, grazie alla collaborazione di un dipendente interno, è riuscito ad ottenere in anticipo le sequenze di gioco previste per le prossime N mani.

Tuttavia, non tutte le mani sono favorevoli. In alcuni casi, nonostante le informazioni, le probabilità restano a vantaggio del banco. Per questo motivo, Daniel si è rivolto a un suo collaboratore esperto di data science, Dammy Dell'Oro, per analizzare i dati storici relativi alle possibili vincite e perdite in fiches per ciascuna mano.

Dammy è riuscito a calcolare, per ogni mano, quanti soldi è possibile vincere o perdere. Per non dare nell'occhio e non suscitare i sospetti dell'azienda, Daniel ha deciso che che si siederà al tavolo, giocherà una serie consecutiva di partite di Black Jack e poi si alzerà. A questo punto, bisogna stabilire quale sia il momento migliore per sedersi e per andarsene dal tavolo, in maniera da massimizzare la vincita in fiches.

Ad esempio, supponiamo che Daniel giochi per 10 mani consecutive ed abbia a disposizione la seguente sequenza di risultati, in termini di fiches vinte o perse:

```
31 -41 59 26 -53 58 97 -93 -23 84
```

Analizzando i dati, si osserva che il miglior risultato si ottiene iniziando dalla terza mano (in cui si possono vincere 59 fiches) e terminando alla settima (in cui si possono vincere 97 fiches), per un guadagno totale di 187 fiches, e non

esiste, con i dati visti in precedenza, una sequenza di mani giocate che riesce a vincere un numero maggiore di fiches.

Il vostro compito è quello di scrivere un programma che, ricevuta in ingresso la lista delle fiches che è possibile vincere (o perdere) in una sequenza di N mani di gioco, restituisca il numero massimo di fiches che è possibile vincere giocando consecutivamente al tavolo, e a quali mani bisogna sedersi ed alzarsi, rispettivamente, dal tavolo.

2 Informazioni ausiliarie

Si tengano in considerazione le seguenti informazioni:

- N, il numero di mani in cui si gioca, è minore di 100000;
- in ogni mano, si possono vincere o perdere al massimo 1000 fiches;
- l'input è costituito dal numero di partite osservate (ad esempio 10) e la lista di vincite/perdite (ad esempio $[31, -41, 59, 26, -53, 58, 97, -93, -23, 84]$
- l'output deve riportare il numero massimo di fiches che è possibile vincere (ad esempio 187), il numero di partita a cui Daniel conviene sedersi (ad esempio 3), il numero di partita a cui Daniel conveiene alzarsi (ad esempio 7), e la sottosequenza delle partite che deve giocare (ad esempio $[59, 26, -53, 58, 97]$).

Per la gestione dell'output usate la seguente classe Java:

```
public class GiocataVincente {
    int start; // Giocata in cui Daniel si siede
    int end; // Giocata in cui Daniel si alza
    int maxWin; // Massima vincita
    int[] subPlay; // Sottosequenza di partite giocate

    public GiocataVincente(int start, int end, int maxWin, int[] subPlay) {
        this.start = start;
        this.end = end;
        this.maxWin = maxWin;
        this.subPlay = subPlay;
    }
}
```

3 Specifiche del codice e tasks

Task 1. Inizialmente, Daniel ha intezione di offrirvi una parcella minima, per cui la soluzione che voi fornirete avrà prestazioni minime. Scrivete il seguente modulo che abbia complessità $O(N^3)$.

```
public static GiocataVincente bruteForce(int[] listaPartite)
```

Task 2. Daniel si rende conto che la soluzione fornita non gli fornisce i risultati in tempo per massimizzare la sua vincita, quindi raddoppia la parcella che deve fornirvi. A quel prezzo, riuscite a fornirgli una soluzione più efficiente. Scrivete il seguente modulo che abbia complessità $O(N^2)$.

```
public static GiocataVincente faster(int[] listaPartite)
```

Task 3. Dammy Dell'Oro inizia a capire il vostro ragionamento e vuole *Divide*rvi da Daniel, a cui fornisce una soluzione con complessità $O(N^2)$ a minor prezzo. Per mantenere Daniel come cliente, dovete dimostrargli di riuscire a fare di meglio in qanto *Impera*tori degli algoritmi. Scrivete il seguente modulo che abbia complessità $O(N \cdot logN)$.

```
public static GiocataVincente bolt(int[] listaPartite)
```

Task 4. Nel giro del gioco d'azzardo si è sparsa ormai la voce delle vostre abilità ed un grande magnate è disposto a ricoprirvi di oro se riuscite a fornirgli la soluzione più efficiente e *dinamica* mai vista. Scrivete il seguente modulo che abbia complessità $O(N)$.

```
public static GiocataVincente goldenSolution(int[] listaPartite)
```

4 Test

Gli algoritmi vanno implementati nella classe *Azzardo.java* con le firme dei moduli riportati nelle specifiche dei tasks.

Per testare il codice compilate ed eseguite la classe Driver. Si possono fare i seguenti tipi di test:

- Testare un singolo algoritmo. In questo caso va eseguito il Driver con l'indicatore dell'algoritmo da eseguire, che può essere: *brute*, *faster*, *bolt*, *golden*, per i 4 tasks rispettivamente. In questo caso viene eseguito il test di esempio riportato nella descrizione del problema.
- Comparare tutti gli algoritmo. In questo caso ca esguito il Driver senza parametri e verranno eseguiti 5 test e comparati i risultati e tempi di esecuzione.

Esempio di compilazione ed esecuzione del Driver per lanciare l'algoritmo di bruteForce:

```
$ javac Driver.java
$ java Driver brute
```

Sostituire *brute* con uno degli altri parametri per i diversi algoritmi.

Esempio di compilazione ed esecuzione del Driver per lanciare tutti i test comparativi:

```
$ javac Driver.java
$ java Driver
```

NB: La classe Driver NON va modificata, ma solo esguita per i test!