# CoderFarm - Corso base

## Lezione 10

Carlo Collodel, Francesco Cerroni

16 gennaio 2023

# Ciao!

Dopo la correzione dei problemi del Round 3 OIS, vediamo i contenuti della scorsa lezione che non abbiamo fatto in tempo a finire!

# Ricorsione

## Chessboard and Queens

```
int solve(int i) {
    if(i >= 8) return 1;

    int ans = 0;

    for(int j = 0; j < 8; ++j) {
        /* provo a piazzare una regina nella posizione (i, j) */
        if(valid(i, j)) {
            rows[j] = true; /* esploro uno stato */
            northe[7 + j - i] = true;
            southe[j + i] = true;

            ans += solve(i + 1);

            rows[j] = false; /* torno indietro (backtracking) */
            northe[7 + j - i] = false;
            southe[j + i] = false;
        }
    }

    return ans;
}
```

# Ricorsione

## Grid Paths

```
int dx[] = {-1, 1, 0, 0};
int dy[] = {0, 0, -1, 1};

int solve(int i, int j, int op) {
    /* ho gia` visitato questa casella */
    if (grid[i][j])
        return 0;

    /* sopra, sotto e` visitato, a destra e sinistra no */
    if ( grid[i + dx[0]][j + dy[0]] &&
         grid[i + dx[1]][j + dy[1]] &&
        !grid[i + dx[2]][j + dy[2]] &&
        !grid[i + dx[3]][j + dy[3]])
        return 0;

    /* destra, sinistra e` visitato, sopra e sotto no */
    if (!grid[i + dx[0]][j + dy[0]] &&
        !grid[i + dx[1]][j + dy[1]] &&
         grid[i + dx[2]][j + dy[2]] &&
         grid[i + dx[3]][j + dy[3]])
        return 0;

    /* sono arrivato alla fine */
    if (i == 7 && j == 1) {
        return (op == 48);
    }
```

# Sorting and Searching

## Restaurant Customers

Vengono dati i tempi di *entrata* e di *uscita* di $N$ clienti in un ristorante, trova il numero massimo di clienti presenti contemporaneamente nel ristorante in un qualsiasi momento.

### Osservazione

Se un cliente esce nello stesso momento in cui entra un altro cliente, conto entrambi i clienti come presenti?
Fortunatamente nella descrizione del problema c'è scritto che "Possiamo assumere che tutti i tempi di entrata e uscita sono distinti".

# Sorting and Searching

## Restaurant Customers

Una prima idea potrebbe essere la seguente:

- Creo un array grande che può contenere il massimo tempo di entrata/uscita possibile (con i valori inizialmente a 0)
- Per ogni cliente, aggiungo 1 nell'array all'indice rispettivo al tempo di entrata, tolgo 1 nella posizione del tempo di uscita
- Quando ho finito, itero per ogni elemento dell'array e mantengo una somma degli elementi considerati fin'ora: ogni volta che aggiungo un elemento alla somma controllo se la mia somma corrente è massima e tengo sempre la più grande.

La complessità è lineare nei tempi di entrata e uscita massimi.

# Sorting and Searching

## Restaurant Customers

### Errori frequenti

L'algoritmo che abbiamo visto è lineare, quindi può sembrare che sia ottimale per il nostro problema, ma se osserviamo i *Constraints* nel testo del problema, possiamo notare che $1 \leq a < b \leq 10^9$ (dati $a, b$ rispettivamente tempo di entrata e uscita).
Quindi il nostro algoritmo è lineare, ma $a$ e $b$ sono troppo grandi per ammettere un algoritmo lineare sulla loro dimensione![1]

Pensiamo a un algoritmo alternativo...

[1] Se invece usassi una `std::map`, questo algoritmo potrebbe funzionare!

# Sorting and Searching

## Restaurant Customers

- Creo un `vector<pair<int, int>>` in cui salvo i tempi di entrata e uscita di ogni cliente, uso il primo `int` per il valore del tempo e nel secondo `int` inserisco 1 o -1 per distinguere un'entrata o un'uscita.
- Ordino il vettore.
- Come prima mantengo una somma iterando per ogni elemento del vettore, sommo 1 in caso di entrata e -1 in caso di uscita (mi basta sommare il secondo campo del `pair`) e poi tengo la somma massima come risposta.

La complessità è $\mathcal{O}(n \log n)$, che è **peggiore** di una complessità lineare, però dai *Constraints* osserviamo che $N \leq 2 \cdot 10^5$: questo algoritmo sta nei tempi di esecuzione!

# Greedy

## Towers

Vengono dati $N$ cubi di dimensioni diverse in un certo ordine. Qual è il numero minimo di *torri* che si possono creare usando questi cubi? Una *torre* è composta da diversi cubi posizionati uno sopra l'altro, ogni cubo deve essere più piccolo del cubo sotto di lui.

# Greedy

## Towers

Quando conviene creare una torre nuova al posto di piazzare un cubo su una torre già esistente?

### Torri nuove?

Supponiamo di dover piazzare un cubo con dimensione $X$ e di avere una torre già costruita il cui cubo in cima ha dimensione $Y$ (con $X < Y$, in altre parole posso piazzare $X$ sopra $Y$).

- Se creo una nuova torre, avrò due torri, con gli elementi in cima rispettivamente $X$ e $Y$.
- Se non creo una nuova torre, avrò una sola torre con l'elemento in cima $X$.

# Greedy

## Towers

Se ho più torri dove posso piazzare un cubo, in quale di queste mi conviene piazzarlo?

### Più torri?

Supponiamo di dover piazzare un cubo con dimensione $X$ e di avere due torri con cubi in cima di dimensioni $Y, Z$ (con $X < Y < Z$).

- Se piazzo $X$ sopra $Y$, ottengo due torri con $X, Z$ in cima.
- Se piazzo $X$ sopra $Z$, ottengo due torri con $X, Y$ in cima.

# Binary Search The Answer

## Array Division

Viene dato un array di $N$ interi positivi.
Dividere l'array in $K$ sottoarray (contigui) in modo che il massimo tra le somme degli elementi di ogni sottoarray è minimo.

### Un problema più semplice

Questo problema sembra molto complicato all'inizio, pensiamo a un problema un po' più semplice!
*Viene dato un array di $N$ interi positivi.*
*Dividere l'array in $K$ sottoarray (contigui) in modo che il massimo tra le somme degli elementi di ogni sottoarray non superi $M$.*

# Binary Search The Answer

## Array Division

Nel caso del problema semplificato, ci basta iterare sull'array e creare dei sottoarray aggiungendo elementi finchè non si sta per superare somma $M$.
Poi conto il numero di sottoarray creati in questo modo:

- Se questi sono più di $K$, allora sicuramente non esiste soluzione a questo problema
- In caso opposto, posso sempre spezzare i sottoarray creati fino ad ottenere una suddivisione in esattamente $K$ sottoarray (se spezzo un sottoarray, ottengo due sottoarray con somma più piccola).

**Attenzione!**

Queste cose valgono solo grazie alla condizione che **ogni elemento è positivo!**

# Binary Search The Answer

## Array Division

**Osservazione**

Se posso costruire un array valido per $M$, quell'array è valido anche per qualsiasi valore $\geq M$.

Allora posso fare **ricerca binaria** sul valore di $M$, trovando il *più piccolo $M$ che ha soluzione*!
Questo tipo di algoritmi si chiama **Binary Search The Answer**: letteralmente si fa ricerca binaria sulla risposta del problema e si controlla se è valida.

# Esercizi per casa!

- Distinct Numbers (difficoltà Facile)
  `https://cses.fi/problemset/task/1621`
- Sum of Two Values (difficoltà Media)
  `https://cses.fi/problemset/task/1640`
- Maximum Subarray Sum (difficoltà Media)
  `https://cses.fi/problemset/task/1643`
- Subarray Distinct Values (difficoltà Difficile)
  `https://cses.fi/problemset/task/2428`

# Fine

Grazie per averci seguito!
Ci vediamo alla prossima lezione :)

- **E-Mail**: `base@coderfarm.it`