

CoderFarm - Corso base

Lezione 8

Carlo Collodel, Francesco Cerroni

8 gennaio 2023

Greedy

Ancora...

Molti problemi delle olimpiadi (specialmente quelli più semplici) si risolvono con tecniche *greedy*!

Problema

https://training.olinfo.it/#/task/oii_orticoltura

Orticoltura

Idea dietro la soluzione

Idea dietro alla soluzione:

- ▶ Ordino i semi secondo la loro posizione
- ▶ Tengo un vector per memorizzare gli irrigatori posizionati
- ▶ Ad ogni iterazione considero l' i -esimo seme e posiziono un irrigatore sopra di lui con tempo di apertura uguale alla profondità
- ▶ Sempre ad ogni iterazione provo a sostituire gli ultimi 2 irrigatori con uno solo finchè mi conviene

Applicando questa idea riesco alla fine del processo a ottenere sia il costo minimo che la posizione degli irrigatori.

Orticoltura

Complessità

Ad ogni iterazione aggiungiamo un irrigatore e ognuno di questi viene al massimo aggiunto e rimosso una volta. In totale eseguiamo circa $2N$ operazioni. Supponendo di riuscire a capire quando conviene sostituire 2 irrigatori in tempo costante ($O(1)$), la complessità finale è $O(N \log N)$ per l'ordinamento iniziale.

Orticoltura

Manca qualcosa?

Ma in pratica come faccio a capire quando conviene sostituire 2 irrigatori con uno soltanto? Esercizio per voi!

Quickselect

Motivazione

Supponiamo ci sia dato un array A e un numero m , vogliamo trovare qual'è l'elemento dell'array che si trova in posizione m quando l'array è ordinato.

Soluzione naïve

Ordino l'array, controllo che valore si trova in posizione m .
Complessità: $\mathcal{O}(N \log N)$.

Quickselect

Motivazione

La soluzione $\mathcal{O}(N \log N)$. non è male di per se, ma comunque in caso di limiti molto stretti potrebbe essere meglio cercare una soluzione più veloce...

Esiste un algoritmo detto **Quickselect**!

I passaggi sono molto semplici...

- ▶ Scelgo un elemento casuale nell'array.
- ▶ Sposto a sinistra tutti gli elementi più piccoli (nell'ordine in cui sono presenti), sposto a destra quelli più grandi.
- ▶ Se l'elemento scelto casualmente si trova in posizione m , ho finito, altrimenti ripeto l'algoritmo nella parte di destra o di sinistra (dipendentemente dalla posizione dell'elemento scelto).

Quickselect

Spiegazione

Se ricordate, un passaggio simile veniva fatto anche nel **Quicksort**, con la scelta del pivot!

Ma analizziamo la complessità di questo algoritmo...

Di media, scegliendo un elemento casuale, spezzeremo circa a metà l'array di partenza.

Dato N il numero di elementi in A , otteniamo un numero di operazioni *attese* pari a $N + N/2 + N/4 + \dots$ (spostiamo sempre tanti elementi quanti quelli dell'intervallo che stiamo considerando).

Si può dimostrare che questa somma è pari a $2N$, la complessità è dunque lineare: $\mathcal{O}(N)$!

Quickselect

Implementazione


In C++ non serve implementare *Quickselect*, esiste già una funzione di libreria (in `#include <algorithm>`) che **modifica** l'ordine dell'array originale, spostando in posizione m l'elemento corrispondente a quella posizione se l'array fosse ordinato.

Con `std::nth_element(A.begin(), A.begin() + m, A.end())`

Il primo e l'ultimo parametro indicano il range dove trovare l'elemento, il secondo parametro indica la posizione (sotto forma di iteratore) di cui si vuole trovare l'elemento corrispondente.

Struct

Le struct sono modi di raggruppare informazioni in un solo tipo:



```
struct Point {  
    int x, y;  
}  
  
Point P = {2, 3};  
P.x = 5;  
cout << P.x << ", " << P.y << "\n"; // 5, 3
```

Struct

Possono anche comparire tipi diversi come campi della struct:

```
struct Persona {  
    int anni;  
    string nome;  
}  
  
Persona Tutor = {18, "Carlo"};  
Tutor.eta = 19;  
Tutor.nome = "Francesco";
```

Struct

Metodi

Le struct possono contenere dei *metodi*, cioè funzioni che possono accedere ai campi della struct:

```
struct Point {  
    int x, y;  
  
    void stampa(string separatore) {  
        cout << x << separatore << y;  
    }  
  
    void scambia_coordinate() {  
        swap(x, y);  
    }  
}
```

```
Point P = {2, 4};  
P.stampa(","); // 2,4  
P.scambia_coordinate();  
P.stampa(","); // 4,2
```

std::pair

std::pair è una struct presente nella stl che contiene 2 campi generici:

```
pair<int, char> coppia = {2, 'a'};  
cout << coppia.first << ", " << coppia.second << "\n"; // 2,a
```

Inoltre le struct implementano di default il seguente ordinamento:

- ▶ Vengono confrontati i primi campi del pair se questi sono diversi
- ▶ Altrimenti vengono confrontati gli altri campi

Esempi: $(1, 1) > (0, 4)$, $(2, 3) < (2, 5)$.

Altre funzioni utili Standard Library

Lista

- ▶ `max_element`
- ▶ `min_element`
- ▶ `maxmin_element`
- ▶ `accumulate`
- ▶ `iota`
- ▶ `swap`
- ▶ `reverse`
- ▶ `unique`
- ▶ `stable_sort`
- ▶ `upper_bound`
- ▶ `lower_bound`
- ▶ `is_sorted`
- ▶ `all_of`

std::swap

std::swap permette di scambiare 2 variabili (dello stesso tipo):



```
int a = 5;  
int b = 2;  
swap(a, b);  
cout << a << " " << b; // 2 5
```


std::iota

std::iota riempie un array con valori che aumentano progressivamente:

```
vector<int> v(5);  
iota(v.begin(), v.end(), 3);  
for (int i: v) {  
    cout << i << " "; // 3 4 5 6 7  
}
```


std::reverse


std::reverse inverte i valori di un container:



```
vector<int> v(5) = {1, 2, 3, 4, 5};  
reverse(v.begin(), v.end());  
for (int i: v) {  
    cout << i << " "; // 5 4 3 2 1  
}
```

std::lower_bound e std::upper_bound

std::lower_bound effettua una ricerca binaria su un container restituendo un puntatore alla primo elemento maggiore o uguale a quello fornito. std::upper_bound restituisce invece un puntatore al primo elemento strettamente maggiore di quello fornito:



```
vector<int> v(5) = {1, 2, 5, 10, 14};  
int pos = lower_bound(v.begin(), v.end(), 5) - v.begin();  
cout << pos; // 2  
pos = upper_bound(v.begin(), v.end(), 5) - v.begin();  
cout << pos; // 3
```

Altre funzioni utili Standard Library

Enumerare le permutazioni

Uso l'istruzione `next_permutation` (in `#include <algorithm>`):

```

#include <algorithm>

...

vector<int> arr = {19, 12, 45};

/* inizialmente ordino l'array */
sort(arr.begin(), arr.end());

do {
    /* faccio qualcosa con la permutazione corrente */
    cout << "{" << arr[0] << ", " << arr[1] << ", " << arr[2] << "}\n";
} while (next_permutation(arr.begin(), arr.end()));

/*
 * next_permutation ritorna true solo quando riesce a costruire una
 * permutazione lessicograficamente piu` grande.
 * quando esco dal ciclo, l'array torna ordinato!
 */
```

Enumerare le permutazioni

Ricorsione

Proviamo a trovare una ricorrenza che ci permetta di esaurire tutte le permutazioni possibili di un array (di dimensione N).

Consideriamo una posizione nell'array: in tutte le permutazioni possibili, **ogni elemento** dovrà finire almeno una volta in quella posizione!

- ▶ Proviamo a fissare un elemento qualsiasi (supponiamo di posizione i) nella posizione 0 della permutazione finale.
- ▶ Ci resteranno esattamente $N - 1$ elementi da permutare nelle posizioni $[1, N - 1]$, ma questo è esattamente un caso più semplice del problema iniziale: *vogliamo enumerare tutte le permutazioni di un array di dimensione $N - 1$!*
- ▶ Quando arriviamo a un array di dimensione 1, l'unica permutazione è l'array stesso.

Enumerare le permutazioni

Implementazione ricorsiva

```
void print_perms(vector<int> &arr, int i) {
    /* quando mi resta un solo elemento, mi fermo */
    if (i == arr.size() - 1) {
        /* modo rapido di stampare un intero array */
        copy(arr.begin(), arr.end(), ostream_iterator<int>(cout, " "));
        cout << endl;
        return;
    }

    /* sposto ogni elemento all'inizio e permuto il resto */
    for (int j = i; j < arr.size(); j++) {
        swap(arr[i], arr[j]);      /* sposto all'inizio */
        print_perms(arr, i + 1);  /* calcolo le permutazioni */
        swap(arr[i], arr[j]);      /* annullo la mia modifica */
    }
}

int main() {
    vector<int> arr = {19, 12, 45};

    /* questa funzione ricorsiva non cambia il mio array alla fine! */
    print_perms(arr, 0);
}
```

Backtracking Ricorsivo

Cos'è?

Il *Recursive Backtracking* è una famiglia di algoritmi che permette di risolvere problemi di *ricerca completa*, *conteggio del numero di configurazioni valide* e molto altro...

Generalmente, si scrive una funzione ricorsiva che prova una configurazione che porta a una soluzione potenziale, ricorre (cercando di trovare una soluzione) e poi esegue il "*backtracking*", cioè **annulla una sua scelta** e ne prova un'altra (per trovare il resto delle soluzioni).

Questa operazione si ripete fino all'esaurimento totale dello spazio di ricerca.

Backtracking Ricorsivo

Esempio

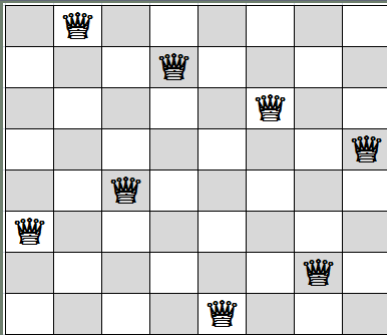
Problema delle regine

Ci viene data una scacchiera $N \times N$, vogliamo contare il numero di modi possibili per piazzare N regine sulla scacchiera in modo che nessuna di queste si "attacchi".

Backtracking Ricorsivo

Esempio

Esempio di configurazione valida



Fonte: codepumpkin.com

Backtracking Ricorsivo

Soluzione

Possiamo usare un algoritmo di *backtracking*!

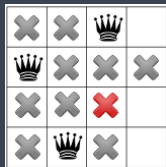
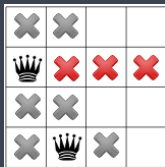
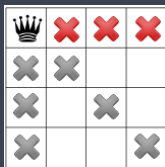
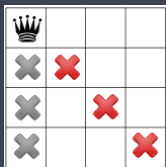
Ricorriamo sulla colonna su cui vogliamo piazzare la prossima regina (ricordiamo che possiamo piazzare una sola regina per colonna) e manteniamo informazioni sulle diagonali e le righe occupate.

Anche qui, proviamo tutte le posizioni nella colonna corrente in cui possiamo piazzare la prossima regina!

Backtracking Ricorsivo

Soluzione

Vediamo alcuni passi (1-2-3-4-6-9-10-12) dell'istanza dell'algoritmo in esecuzione:



Fonte: <https://developers.google.com/optimization/cp/queens>

Backtracking Ricorsivo

Complessità

Calcoliamo la complessità di quest'algoritmo!

Se chiamassi $T(N)$ il tempo necessario per risolvere un'istanza del problema su una scacchiera $N \times N$, avrei:

$$T(N) = N \cdot T(N - 1) + O(N^2)^1$$

Questo si traduce in $O(N!)$, perché $N! = N \cdot (N - 1) \cdots 1$.

Possiamo concludere che quest'algoritmo è molto lento, infatti molto spesso la ricerca con backtracking porta ad algoritmi con complessità esponenziale!

Il problema della complessità di questi algoritmi, spesso viene risolto con due tecniche: **Programmazione dinamica** (che vedremo prossimamente), **Branch and Bound** e **ottimizzazioni generali**.

¹ N^2 è la complessità necessaria per controllare se è possibile scegliere una determinata posizione per ogni casella della colonna

Branch and Bound

Idea generale

L'idea del **Branch and Bound** è di ottimizzare una ricerca esaustiva "tagliando" i rami di ricorsione quando si è sicuri di non star raggiungendo una soluzione ottimale.

Un modo di fare ciò consiste nel memorizzare la migliore soluzione trovata al momento e **uscire** da tutte le successive chiamate ricorsive che superano quel valore come soluzione corrente (anche prima di arrivare a una soluzione completa, se possibile).

Branch and Bound

Idea generale

In caso non avessimo nessun "punteggio" con cui possiamo ridurre le chiamate ricorsive, gli algoritmi **BnB** degenererebbero a ricerche esaustive (come la soluzione del Problema delle Regine).

Un modo per aumentare l'efficacia di questi algoritmi è **dare un ordinamento** alle scelte che si possono fare usando un'*euristica*.

Problema dello zaino (variante 0/1)

Ho uno zaino di capacità C_c , ci sono N oggetti rispettivamente di valore V_i e di peso W_i .

Trovare il **sottoinsieme** di oggetti con peso totale $\sum W_i \leq C$ tale che il loro valore $\sum V_i$ sia massimale.

Knapsack

Bruteforce ricorsivo

La soluzione ricorsiva consiste nel provare tutti i sottoinsiemi possibili, calcolare le somme e tenere la soluzione valida con valore massimo.

Complessità: $\mathcal{O}(2^N)$ (provo tutti i sottoinsiemi possibili).

Esercizio per voi: provate a scriverla *iterativa* e anche *ricorsiva*!

Knapsack

Branch and Bound

Possiamo modificare la soluzione ricorsiva mantenendo una variabile globale **costo** e una variabile globale **guadagno**, aggiornandole ad ogni scelta presa.

Posso ottimizzare la mia ricerca "ritornando" dalle chiamate ricorsive appena il costo supera la capacità totale.

Un'altra ottimizzazione che posso fare è mantenere un'ulteriore variabile globale **valore ottimale** e controllare se "*costo*" + "*valore totale degli elementi che posso ancora prendere*" è inferiore al "*valore ottimale*", in quel caso si può ritornare dalla chiamata ricorsiva.

Come ultima ottimizzazione, potrei ordinare gli elementi per favorire il "pruning" della ricerca.

Ottimizzazioni

Idea generale

Nella ricerca ricorsiva possiamo fare molte ottimizzazioni che possono risparmiare una grande parte del tempo di esecuzione.

Grid Paths

Contare il numero di percorsi in una griglia $N \times N$ che partono dall'angolo in alto a sinistra della griglia e finiscono nell'angolo in basso a sinistra.

Grid Paths

Suggerimento

Non vi daremo la soluzione finale, però un suggerimento importante:

Ci sono diversi modi di fare "pruning" su questa ricerca completa, ma ce n'è uno estremamente efficace!

Pensate al caso in cui ottenete una "T" mentre vi state muovendo: data (i, j) la vostra posizione (i indica la riga, j la colonna) e la vostra direzione è per esempio verso il basso: $(i + 1, j)$ è occupata e $(i, j + 1)$, $(i, j - 1)$ sono invece libere.

Esercizi per casa!

- ▶ Chessboard and Queens

<https://cses.fi/problemset/task/1624>

- ▶ Grid Paths

<https://cses.fi/problemset/task/1625>

- ▶ Numero della cabala

https://training.olinfo.it/#/task/ois_cabala

- ▶ Gioco del tris

https://training.olinfo.it/#/task/ois_tris/statement

Fine

Buone vacanze!

- ▶ **E-Mail:** `base@coderfarm.it`
- ▶ **Discord:** T.B.D.