CoderFarm - Corso base

Lezione 6

Carlo Collodel, Francesco Cerroni

15 dicembre 2022

RMI2020 Floppy

Cominciamo con un problema!

Testo del problema

Viene dato un array A di lunghezza N. Bisogna "salvare" dei dati in base agli elementi dell'array in modo tale da riuscire a rispondere a delle query del tipo: "qual è il valore massimo A_i dell'array nel range $[L; R]$?".
Dal numero di bit salvati per risolvere il problema, il punteggio del problema viene assegnato diversamente: per ottenere il massimo non bisogna superare $2N$ bits utilizzati.

RMI2020 Floppy

Esempio e suggerimenti

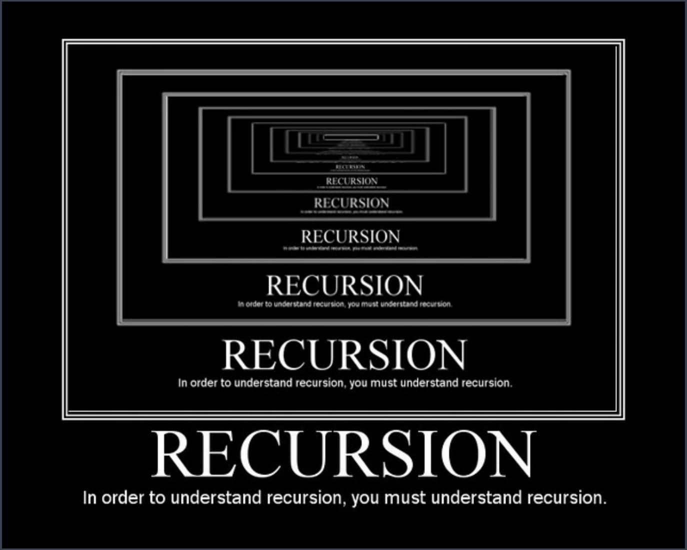
RMI2020 Floppy

Vediamo la soluzione!

Soluzione

Nearest smaller element!
Salvo con 0 `push` e con 1 `pop`, in questo modo posso riprodurre l'intero algoritmo.
Poi parto da un estremo, continuo a seguire la posizione dello smaller element fino a superare il prossimo estremo.
Provate a implementarlo! (potete consegnare il problema al link: `https://oj.uz/problem/view/RMI20_floppy`)

Ricorsione


RECURSION
In order to understand recursion, you must understand recursion.
RECURSION
In order to understand recursion, you must understand recursion.
RECURSION
In order to understand recursion, you must understand recursion.
RECURSION
RECURSION
RECURSION

Ricorsione

Definizione

In informatica (e in altre discipline) parliamo di ricorsione quando possiamo **definire qualcosa tramite se stessa**.

Esempio

Il fattoriale è definito come $N! = N \cdot (N-1) \cdots 2 \cdot 1$
Possiamo anche definire $N!$ come $N \cdot (N-1)!$
$\rightarrow$ Definizione ricorsiva!
Devo anche definire un *caso base*: $0! = 1$, se non lo facessi, la definizione ricorsiva di fattoriale non avrebbe senso!

Recall: funzioni ricorsive

Fattoriale in C++

```cpp
int fattoriale(int n) {
    /* Caso base */
    if (n == 0)
        return 1;

    /* Passo ricorsivo, una funzione ricorsiva "chiama" se stessa */
    return fattoriale(n - 1) * n;
}
```

Non è sempre facile calcolare la complessità di una chiamata a una funzione ricorsiva.
Nel caso del fattoriale: ho in totale $N + 1$ chiamate alla funzione *fattoriale* (N, $N - 1$, ..., 0), ogni chiamata ha complessità $\mathcal{O}(1)$.
Complessità totale $\to \mathcal{O}(N)$.
Vediamo un altro esempio:

Numeri di Fibonacci

Definiamo l'i-esimo *numero di Fibonacci* come:

$$F_i = \begin{cases} 0 & \text{se } i = 0 \\ 1 & \text{se } i = 1 \\ F_{i-1} + F_{i-2} & \text{se } i > 1 \end{cases}$$

Ricorsione

VS Iterazione

```cpp
/** Implementazione Iterativa **/
int a = 0, b = 1;
for (int i = 2; i <= n; ++i) {
    swap(a, b); /* a diventa il piu` grande */
    b += a;     /* calcolo il numero successivo */
}

cout << (n == 0 ? 1 : b) << endl;

/** Implementazione Ricorsiva **/
int fibonacci(int n) {
    /* Casi base */
    if (n <= 1)
        return n;

    /* Passo ricorsivo, calcolo ricorsivamente i due numeri che mi servono */
    return fibonacci(n - 1) + fibonacci(n - 2);
}

cout << fibonacci(n) << endl;
```

Ricorsione

VS Iterazione

Nell'implementazione iterativa, la complessità è chiaramente $\mathcal{O}(N)$, ho un ciclo che itera $\approx N$ volte.
Quindi anche la complessità dell'implementazione con la funzione ricorsiva sarà $\mathcal{O}(N)$?

Proviamo ad eseguirli!

Eseguo i due programmi e misuro il tempo...

```
❯ time ./iter <<< "45"
1134903170
./iter <<< "45"  0.00s user 0.00s system 85% cpu 0.003 total
❯ time ./rec <<< "45"
1134903170
./rec <<< "45"  6.43s user 0.00s system 99% cpu 6.436 total
```

Ricorsione

Attenzione alla complessità computazionale

L'implementazione ricorsiva **NON** ha complessità $\mathcal{O}(N)$.

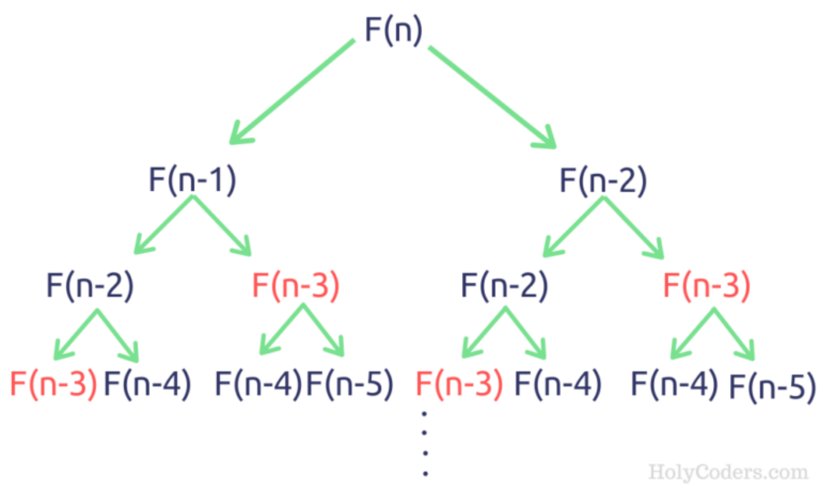
- Ogni chiamata a *fibonacci* ha complessità $\mathcal{O}(1)$ (non considerando le altre chiamate).
- Ogni chiamata effettua **due chiamate** ricorsive!

Definita T la complessità di F:
La complessità è data da: $T(n) = T(n-1) + T(n-2) + \mathcal{O}(1)$.
Si può risolvere la ricorrenza[1] e ottenere $T(n) = \mathcal{O}(\phi^n)$
($\phi = \frac{(1+\sqrt{5})}{2}$, il rapporto aureo).

[1] https://cs.stackexchange.com/questions/14733/complexity-of-recursive-fibonacci-algorithm

Ricorsione

Albero di ricorsione



Divide et impera

(o Divide and conquer)

Esempio fondamentale - Merge sort

Supponiamo di avere due array A, B *ordinati* con $\frac{N}{2}$ elementi.
Possiamo unire i due array in uno unico array C *sempre ordinato* di lunghezza N?
Se sì, qual è la complessità?

Divide et impera

(o Divide and conquer)

Esempio fondamentale - Merge sort

Parto dall'inizio con due indici $i = j = 0$, inserisco nel nuovo array C l'elemento più piccolo tra A_i e B_j. Quando $i == \frac{N}{2}$ oppure $j == \frac{N}{2}$, aggiungo gli elementi rimanenti in coda a C.

Divide et impera

(o Divide and conquer)

Esempio fondamentale - Merge sort

E se avessi due array ordinati di lunghezza $\frac{N}{4}$, potrei unirli per formarne uno ordinato di lunghezza $\frac{N}{2}$?
Posso ripetere la stessa domanda con $\frac{N}{2^{K+1}}$ e $\frac{N}{2^K}$ e così via...

Divide et impera

(o Divide and conquer)

Esempio fondamentale - Merge sort

Ora posso costruire un algoritmo efficiente per ordinare un vettore di dimensione N:

Algorithm 1 Merge Sort - Complessità $\mathcal{O}(N \log N)$

```
function MERGE_SORT(V: vettore di lunghezza N)
    if N == 1 then return V
    sinistra ← MERGE_SORT(V[:N/2])                    ▷ Ordino una metà
    destra ← MERGE_SORT(V[N/2:])                      ▷ Ordino il resto
    risultato ← UNISCI_ORDINATO(sinistra, destra)
    return risultato
```

Divide et impera

(o Divide and conquer)

Il **Divide et impera** è una tecnica di programmazione che permette di dividere un problema in sottoproblemi più semplici fino ad arrivare a dei casi banali (nel caso dell'ordinamento, un vettore di lunghezza 1 è banale da ordinare).
Dopo aver risolto i sottoproblemi, si uniscono per costruire le soluzioni dei problemi più grandi.

Analisi della complessità del Merge Sort

- Ogni chiamata a *merge_sort* effettua due chiamate ricorsive **con vettori di lunghezza dimezzata!**
- Ogni chiamata effettua singolarmente $\mathcal{O}(N)$ (con N lunghezza del vettore con cui è stata chiamata) operazioni.
- la complessità è pari a $\mathcal{O}(N \log N)$:
  $\mathcal{O}(N) + \mathcal{O}(\frac{N}{2}) + \mathcal{O}(\frac{N}{2}) + 4 \cdot \mathcal{O}(\frac{N}{4}) + \cdots + 2^{\log N} \cdot \mathcal{O}(\frac{N}{2^{\log N}})$

Merge sort

Implementazione

```cpp
void merge(vector<int> &v, vector<int> &buffer, int l, int r) {
    int m = (l + r) / 2;
    int i = l, j = m + 1, k = l;
    while (i <= m && j <= r) {
        if (v[i] < v[j]) {
            buffer[k] = v[i];
            i++;
        } else {
            buffer[k] = v[j];
            j++;
        }
        k++;
    }

    for (; i <= m; i++, k++) {
        buffer[k] = v[i];
    }
    for (; j <= r; j++, k++) {
        buffer[k] = v[j];
    }

    for (i = l; i <= r; i++) {
        v[i] = buffer[i];
    }
}
```

Merge sort

Implementazione

```
void recursion(vector<int> &v, vector<int> &buffer, int l, int r) {
    // [l,r] è l'intervallo considerato
    if (r - l < 1) {
        return;
    } else {
        int m = (l + r) / 2;
        recursion(v, buffer, l, m);
        recursion(v, buffer, m+1, r);
        merge(v, buffer, l, r);
    }
}

void mergesort(vector<int> &v) {
    vector<int> buffer(v.size());
    recursion(v, buffer, 0, v.size() - 1);
}
```

Divide et impera

Numero minino di swap

Problema

Dato un array, determinare il numero minimo di operazioni necessarie per ordinarlo. L'unica operazione permessa è scambiare 2 elementi **adiacenti** (trasposizione elementare).

Divide et impera

Numero minimo di swap

Osservazione: il numero minimo di swap necessari è uguale al numero di coppie (i, j) tali che $(a_i > a_j)$ (cioè il numero di coppie non ordinate). Una possibile dimostrazione[2] consiste nell' osservare che se ci sono esattamente k coppie non ordinate sono necessari **almeno** k scambi. Inoltre scambiando 2 elementi adiacenti (con valore diverso) il numero totale di coppie non ordinate aumenta o diminuisce sempre di 1 (provate a dimostrarlo voi). Perciò il numero di scambi necessari è proprio k.

[2] Per approfondire il problema https://codeforces.com/blog/entry/92130

Divide et impera

Numero minino di swap

Questo problema si può risolvere utilizzando il merge sort. Infatti:

- Se ho 2 array ordinati posso facilmente contare il numero di coppie non ordinate durante il merge (vediamo dopo come)
- Nel caso in cui l'array considerato ha lunghezza ≤ 1 allora non ci sono coppie non ordinate

Quindi basta applicare il merge sort aggiungendo la conta delle coppie non ordinate.

Divide et impera

Numero minimo di swap

Per contare il numero di coppie non ordinate durante il merge basta osservare che nel momento in cui l'elemento considerato nel secondo array è minore di quello nel primo, segue che tutti gli elementi ancora non processati del primo array sono maggiori di quello nel secondo. Il valore finale è dato dalla somma delle coppie contate nel primo array, di quelle del secondo array e di quelle contate durante il merge che convolgono elementi da entrambi gli array.

Divide et impera

Numero minimo di swap

Esempio di merge con conta delle coppie:

$$A = [1, 3, 7, 9, 11] B = [2, 4, 5, 6, 10]$$

Esercizi!

- Tornello olimpico `https://territoriali.olinfo.it/#/task/tornello`
- Ordinamento a paletta `https://training.olinfo.it/#/task/oii_paletta/statement`

Fine

Ci vediamo alla prossima lezione!

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