Competitive Programming

Davide Cazzin

Novembre 2021

1 Complessità

Un concetto molto importante nel competitive programming è quello della complessità di un algoritmo. Gli algoritmi che scriviamo per risolvere devono essere efficienti sia in termini di tempo che di spazio. Spesso i problemi che dovremo risolvere ci daranno un indicazione sulla dimensione dei dati in input, il tempo massimo per il quale possiamo eseguire il nostro programma e la memoria massima che può utilizzare.

Vediamo 3 classi di complessità, big O, big theta (Θ) e big omega (Ω).

- Big O: Complessità asintotica, complessità dell'algoritmo nel peggior caso possibile. Definisce un limite superiore sulla complessità dell'algoritmo.
- Big theta: Se big O e big omega sono uguali allora si può dire che big theta è uguale a big O e big omega.
- Big omega: Definisce un limite inferiore sulla complessità dell'algoritmo.

Le classi di complessità possono essere utilizzate per descrivere sia la complessità temporale che la complessità spaziale di un algoritmo.

2 Esempio complessità

Vediamo ora un esempio di complessità di un algoritmo. Supponiamo di dover scrivere l'algoritmo Bubble Sort per ordinare gli elementi di un vettore in ordine crescente.

Listing 1: Bubble Sort in C++

```
#include<iostream>
#include<vector>

using namespace std;

void bubbleSort(vector<int>& v){
    bool ordered = false;
    unsigned int i = 0;
    while(i < v.size() && !ordered){
        ordered = true;
        for(unsigned int j = 0; j < v.size() - i - 1; j++){
            if(v[j] > v[j + 1]){
                int temp = v[j];
                v[j] = v[j + 1];
                v[j + 1] = temp;
```

```
                ordered = false;
            }
        }
        i++;
    }
}

int main(){
    vector<int> v = {1,5,3,2,7,4,6};
    bubbleSort(v);
    for(const int& el : v){
        cout << el << "\n";
    }
}
```

La compessità temporale di questo algoritmo è $T(n) = n * \frac{n}{2}$ quindi diremo che la classe di complessità è $O(n^2)$, il peggior caso si verifica quando il vettore è ordinato con ordine decrescente. Il miglior caso si verifica quando il vettore è già ordinato in ordine crescente, se tale condizione si verifica il vettore verrà visitato una sola volta e poi l'algoritmo terminerà, possiamo quindi dire che $\Omega(n)$.

3 Sort

In competitive programming molto difficilmente dovremmo scrivere un algoritmo di sorting, la stra grande maggioranza delle volte useremo il sort che ci viene fornito dalla standard library.

```
void sort (RandomAccessIterator first, RandomAccessIterator last);

void sort (RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

Il sort utilizzato da questa funzione è chiamato IntroSort e ha una complessità temporale $O(nlogn)$.

Il comparator di default che viene utilizzato dalla funzione sort nel caso non ne venga fornito un altro è $less < int >$ e ordina la nostra struttura dati in ordine non decrescente. Per ordinare il vettore con ordine non crescente possiamo usare il comparator $greater < int >$, come nel seguente esempio:

```
vector<int> v = {1,4,5,6,3,2};
sort(v.begin(), v.end(), greater<int>);
```

Nel caso avessimo un vettore contantente un tipo diverso dagli interi, ad esempio $pair < string, int >$ e vogliamo ordinare questo vettore secondo un ordine che definiamo noi. Vogliamo che il vettore di coppie sia ordinato in base alla stringa in modo crescente e nel caso di stringhe uguali la coppia con l'intero più basso deve precedere.

Listing 2: Custom Comparator

```
/*
    Data una lista di cognomi di studenti e la loro relativa media,
    ordinali con ordine non crescente in base alla loro media,
    in caso di parità ordinali in ordine alfabetico in base al cognome.

    Input:
    n (numero di studenti)
    [cognome] [media]
```

```
    ...

    Output:
    [cognome] [media]
    ...

    Esempio di input:
    4
    Verdi 19
    Bianchi 27
    Rossi 28
    Gialli 27

    Esempio di output:
    Rossi 28
    Gialli 27
    Bianchi 27
    Verdi 19
*/

#include<iostream>
#include<vector>
#include<algorithm>

using namespace std;

bool sortByAscendingValueAndDescendingKey(const pair<string,int>& a, const pair<string,int>& b){
    if(a.second == b.second) return a.first > b.first;
    return a.second > b.second;
}

int main(){
    vector<pair<string,int>> v;
    int n;
    cin >> n;
    string name;
    int avg;
    while(n--){
        cin >> name >> avg;
        v.push_back({name, avg});
    }
    sort(v.begin(), v.end(), sortByAscendingValueAndDescendingKey);
    for(const pair<string,int>& el : v){
        cout << el.first << ' ' << el.second << "\n";
    }
}
```

4 Makefile

Listing 3: Makefile example

```
.PHONY: release

release: SortPairs.cpp
	g++ -Wall -O3 -std=c++17 --pedantic -o SortPairs sortPairs.cpp

debug: SortPairs.cpp
	g++ -Wall -g -std=c++17 --pedantic -o SortPairsDebug sortPairs.cpp

clean:
	rm -f SortPairs SortPairsDebug
```