CoderFarm - Corso base

Lezione 11

Carlo Collodel, Francesco Cerroni

30 gennaio 2023

Subarray Distinct Values

Soluzione

Testo[1]: *Dato un array di N interi, calcola il numero totale di sottoarray (contigui) con al massimo K valori distinti all'interno.*

Prima osservazione

Se ho un sottoarray valido rappresentato da un intervallo $[L, R]$, allora anche tutti i sottointervalli contenuti in $[L, R]$ sono a loro volta sottoarray validi.

[1] https://cses.fi/problemset/task/2428

Subarray Distinct Values

Idea Iniziale

Come prima idea, potrei pensare a un algoritmo *Sliding Window*:

- Uso una `std::map<int, int>` per memorizzare gli elementi nella mia *Window* e le loro rispettive occorrenze.
- Quando espando la *Window* a destra, aumento il valore rispettivo all'elemento aggiunto nella *map*.
- Restringo la *Window* se $map.size() > k$.
- Ottenuto un range di elementi validi, aggiungo al totale delle soluzioni il numero di subrange possibili che posso creare partendo dal range corrente: questi sono $\frac{N(N-1)}{2}$.

Subarray Distinct Values

Idea Corretta

Problema

Se ho due range validi vicini, questa soluzione conta due volte qualsiasi soluzione sia in comune tra quei due range.

Posso risolvere questo problema considerando ad ogni range valido solo **i sottorange validi che terminano nella posizione dell'indice destro**.
Questi intervalli validi sono esattamente pari al numero di elementi correntemente nella *Window*.

Subarray Distinct Values

Codice

```cpp
#include <iostream>
#include <vector>
#include <map>
using namespace std;

int main() {
    int n, k; cin >> n >> k;
    vector<int> a(n);
    for(auto &x : a) cin >> x;

    int i = 0;
    map<int, int> window;
    long long ans = 0ll;
    for(int j = 0; j < n; ++j) {
        /* inserisco l'elemento in posizione j */
        window[a[j]]++;

        while(window.size() > k) {
            auto it = window.find(a[i++]);
            it->second--;
            if(it->second == 0)
                window.erase(it);
        }

        /* aggiungo un valore pari alnumero di elementi nel range */
        ans += j-i+1;
    }

    cout << ans << '\n';
}
```

Programmazione dinamica

Esempio fondamentale

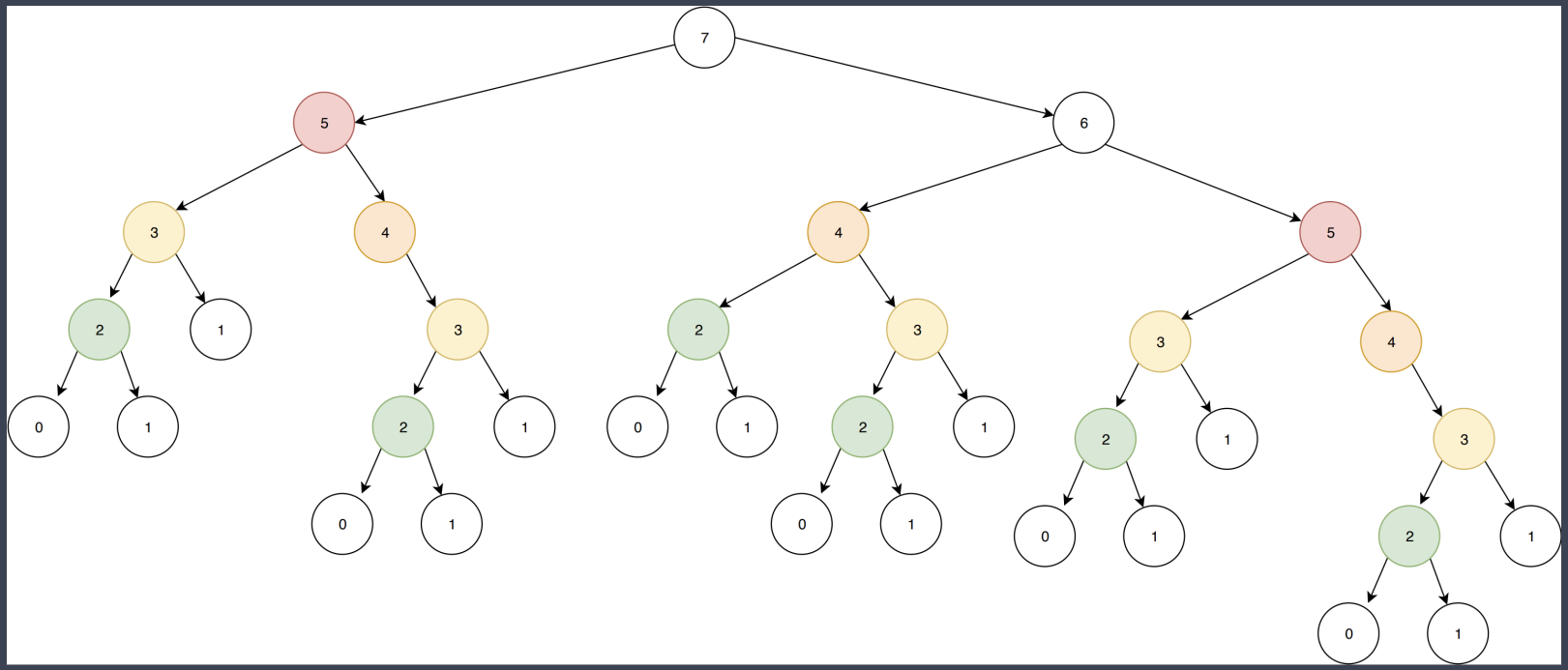
Oggi esploriamo una nuova tecnica molto potente, chiamata programmazione dinamica.

Fibonacci

Riprendiamo il problema di trovare l'N-esimo numero di fibonacci, abbiamo visto nelle scorse lezioni due implementazioni, ricorsiva e iterativa.
L'implementazione iterativa aveva complessità $O(N)$.
L'implementazione ricorsiva aveva complessità $\mathcal{O}(\phi^n)$ (ogni chiamata alla funzione eseguiva *due* chiamate alla funzione).

Programmazione dinamica

Albero di ricorsione



Osserviamo bene l'immagine, alcune chiamate ricorsive vengono **eseguite più volte**, ad esempio $F(3)$ viene chiamato 5 volte partendo da $F(7)$!

Fonte immagine: `guillaumebogard.dev`

Programmazione dinamica

Memoizzazione

Ma allora, dato che $F(n)$ fornisce un risultato sempre uguale dato lo stesso N, non ho bisogno di eseguire la funzione più di **una** volta: Qui entra in gioco la memoizzazione: ogni volta che finisco una chiamata ricorsiva, **salvo** il valore che ho calcolato, al fine di non dover mai più ricalcolare il risultato della funzione con gli stessi parametri.

Complessità computazionale?

Qual è la complessità di un algoritmo ricorsivo con *memoizzazione*? Eseguo 2 operazioni per chiamata ricorsiva, eseguo al massimo N funzioni ricorsive (senza contare quelle in cui riuso il risultato memoizzato): complessità pari a $\mathcal{O}(N)$.

La dieta di Poldo[2]

Il problema

Questa aggiunta al nostro algoritmo può sembrare un po' banale, quindi vediamo un esempio di problema più difficile dove applichiamo la Programmazione dinamica!

Testo

Vengono dati N panini **in ordine** con i loro rispettivi pesi. Un panino può essere mangiato solo quando servito la prima volta rispettando l'ordine con cui vengono serviti.
Un panino può essere mangiato se inoltre rispetta queste condizioni:

- Il panino è il primo ad essere mangiato.
- Il peso dell'ultimo panino mangiato è maggiore del peso del panino servito.

Trovare il numero massimo di panini che è possibile mangiare.

[2] `https://training.olinfo.it/#/task/poldo`

La dieta di Poldo

Soluzione

Soluzione

Spezziamo il problema in N *sottoproblemi* più semplici: qual è il numero massimo di panini che Poldo potrebbe mangiare se mangiasse come primo panino quello che si trova all'indice i? Ragioniamo dal fondo:

- $i = N - 1$ (ultimo elemento): ho solo 1 panino, risposta $\rightarrow 1$.
- $i = N - 2$: mangio l'i-esimo panino, se il peso del panino mangiato è maggiore del peso dell'ultimo panino, mangio anche quello, risposta $\rightarrow 1 +$ [sottoproblema($N - 1$) se peso($N - 2$) $>$ peso($N - 1$)].
- ...

La dieta di Poldo

Soluzione

Soluzione

In generale posso "collegare" i miei sottoproblemi con delle **transizioni** nel seguente modo:

$$dp(i) = \begin{cases} 1 & \text{se } i = N-1 \\ 1 + \max(dp(j))\ \forall j > i,\ peso(i) > peso(j) & \text{se } i < N-1 \end{cases}$$

Con questa ricorrenza stiamo dicendo: "mangio il panino alla posizione i e provo a vedere **qual è il prossimo panino che mi conviene mangiare per ottenere la risposta ottimale**".

Complessità computazionale

Ho N stati diversi, in ogni stato eseguo $\mathcal{O}(N)$ operazioni (provo tutti i $j > i$). Complessità $\mathcal{O}(N \cdot N) = \mathcal{O}(N^2)$.

La dieta di Poldo

Codice ricorsivo

```
int solve(int i) {
    if (i == n-1)
        return 1;

    int& ans = dp[i];
    /* posso sempre mangiare almeno 1 panino */
    if (ans != 0)
        return ans;

    ans = 1;
    for (int j = i+1; j < n; ++j)
        if (peso[i] > peso[j])
            ans = max(ans, 1 + solve(j));

    return ans;
}
```

Poi prendo il massimo tra $solve(0), solve(1), \ldots, solve(N-1)$

La dieta di Poldo

Codice iterativo

```cpp
vector<int> dp(N, 0);
for (int i = N - 1; i >= 0; i--) {
    int migliore = 0;
    for (int j = i + 1; j < N; j++) {
        if (peso[i] > peso[j]) {
            migliore = max(migliore, dp[j]);
        }
    }
    dp[i] = migliore + 1;
}

cout << *max_element(dp.begin(), dp.end());
```

Programmazione dinamica

Tecniche

In generale si possono quindi distinguere 2 tecniche diverse:

- *top-down*: quando si scompone un problema in sottoproblemi partendo dal caso più grande fino ad arrivare ai casi più piccoli (e più semplici).
- *bottom-up*: quando si risolve il problema partendo da sottoproblemi più piccoli (e più semplici), unendoli per formare soluzioni a casi più complicati.

Il secondo è generalmente più veloce anche se in qualche caso potrebbe capitare di avere pochi stati da calcolare in uno "spazio" di ricerca molto grande, sarà quindi più efficiente il primo metodo.

Programmazione dinamica

Tecniche

Le soluzioni in programmazione dinamica si possono scrivere in due diversi modi:

- *implementazione ricorsiva*: trovo una ricorrenza che scompone in sottoproblemi, la implemento in codice e aggiungo **memoizzazione** (spesso *top-down*).
- *implementazione iterativa*: scrivo la soluzione facendo uso di cicli *for*, spesso calcolando i sottoproblemi da più facile a più difficile (*bottom-up*).

L'implementazione iterativa è spesso più veloce e spesso si possono attuare ottimizzazioni di memoria (raramente richieste).
Detto ciò, nella maggior parte dei casi entrambi i tipi di soluzione andranno bene.

Minimizing coins[3]

Problema

Considerate un sistema monetario da N monete con valori $C_i > 0$.
Trovare il numero minimo di monete possibili per ottenere una somma di denaro pari a x.
Constraints: $N \leq 100, C_i \leq 1e6, x \leq 1e6$

[3] https://cses.fi/problemset/task/1634

Minimizing coins

Greedy

Perché non usare una soluzione greedy?
Prendo sempre la moneta più grande, come quando abbiamo risolto l'esercizio con le monete da $1, 2, 5, ...$
Questo non funziona, ho un controesempio con $C = \{1, 4, 5, 7\}$, $X = 9$:

- *Soluzione greedy:* $\{7, 1, 1\}$ (3 monete).
- *Soluzione ottimale:* $\{4, 5\}$ (2 monete).

Minimizing coins

DP

Soluzione bottom-up

Per ottenere 0 come somma, mi servono 0 monete. Per ottenere C_i come somma, mi basta 1 moneta (quella rispettiva al valore). Ora posso iterare per ogni valore di somma già raggiunta (dal più piccolo al più grande) e provare ad "aggiungere" ogni tipo di moneta, vedendo se ottengo un valore nuovo.

Minimizing coins

Codice

```cpp
sort(c.begin(), c.end());

vector<int> dp(x + 1, INF);
dp[0] = 0;
for(int i = 1; i <= x; ++i) {
    for(auto &v : c) {
        if(i - v >= 0) {
            dp[i] = min(dp[i], dp[i - v] + 1);
        } else {
            break;
        }
    }
}
```

Esercizi per casa!

- Dice Combinations (difficoltà facile)
  `https://cses.fi/problemset/task/1633`
- Coin Combinations I (difficoltà facile)
  `https://cses.fi/problemset/task/1635`
- Grid Paths (difficoltà media)
  `https://cses.fi/problemset/task/1638`
- Coin Combinations II (difficoltà difficile)
  `https://cses.fi/problemset/task/1636`

Fine

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

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