Stringhe, rolling hash

e un po' di matematica

Lorenzo Ferrari, Davide Bartoli

May 3, 2023

Table of contents

Fast Modular Exponentation

Exponentation

Date $n \leq 2 \cdot 10^5$ coppie a, b, calcola efficientemente a^b modulo $10^9 + 7$.
https://cses.fi/problemset/task/1095/

Fast Modular Exponentation

Exponentation

Date $n \leq 2 \cdot 10^5$ coppie a, b, calcola efficientemente a^b modulo $10^9 + 7$.
https://cses.fi/problemset/task/1095/

- soluzione naive: b moltiplicazioni, $O(b)$

Fast Modular Exponentation

Exponentation

Date $n \leq 2 \cdot 10^5$ coppie a, b, calcola efficientemente a^b modulo $10^9 + 7$.
https://cses.fi/problemset/task/1095/

- soluzione naive: b moltiplicazioni, $O(b)$
- soluzione ottima: $O(\log b)$ usando la seguente ricorsione

$$a^b = \begin{cases} 1 & \text{se } b = 1 \\ \left(a^{\frac{b}{2}}\right)^2 & \text{se } b \text{ pari} \\ \left(a^{\frac{b-1}{2}}\right)^2 \cdot b & \text{se } b \text{ dispari} \end{cases}$$

Fast Modular Exponentation

Implementazione ricorsiva

```
long long fxp(long long a, long long b) {
    if (b == 0) return 1;
    long long ans = fxp(a, b / 2);
    ans = ans * ans % mod;
    if (b % 2 == 1) return ans * a % mod;
    else return ans;
}
```

Fast Modular Exponentation

Implementazione iterativa

```
long long fxp(long long x, long long y) {
    long long ans = 1;
    while (y) {
        if (y & 1) {
            ans = ans * x % mod;
        }
        y >>= 1;
        x = x * x % mod;
    }
    return ans;
}
```

Aritmetica modulare

Sia m un intero positivo detto **modulo**. Gli interi $0, \ldots, m-1$ prendono il nome di *classi di resto* modulo m.

- due interi a, b appartengono alla stessa classe di resto r se entrambi danno resto r nella divisione intera con m
- due interi appartenenti alla stessa classe di resto si dicono *congrui* modulo m:

$$a \equiv b \pmod{m} \iff m|(a-b)$$

[1] *quasi* vero: `a % mod` con `a` negativo restituisce un intero in $[m-1, 0]$

Aritmetica modulare

Sia m un intero positivo detto **modulo**. Gli interi $0, \dots, m-1$ prendono il nome di *classi di resto* modulo m.

- due interi a, b appartengono alla stessa classe di resto r se entrambi danno resto r nella divisione intera con m
- due interi appartenenti alla stessa classe di resto si dicono *congrui* modulo m:

$$a \equiv b \pmod{m} \iff m | (a - b)$$

- in C++, l'operatore che associa un intero alla sua classe di resto[1] modulo `mod` è "%", `12 % 7 == 5`

Addizione, sottrazione e moltiplicazione funzionano normalmente anche sotto modulo (la divisione no!)

[1] *quasi* vero: `a % mod` con `a` negativo restituisce un intero in $[m-1, 0]$

Numeri coprimi

Phi di Eulero

Due interi a, b si dicono *coprimi*[2] se il loro massimo comun divisore (gcd) è 1.

- 1 è coprimo con tutti
- 0 non è coprimo con nessun intero ≥ 2

[2] o *primi tra loro*
[3] letto "*phi* di *n*"

Numeri coprimi

Phi di Eulero

Due interi a, b si dicono *coprimi*[2] se il loro massimo comun divisore (gcd) è 1.

- 1 è coprimo con tutti
- 0 non è coprimo con nessun intero ≥ 2

Sia $\phi(n)$[3] il numero di interi coprimi con n nel range $[1, n]$.

- $\phi(p) = p - 1$ per ogni p primo
- in generale, $\phi(p^k) = (p-1)p^{k-1}\ \forall k \in \mathbb{Z}_+$
- ϕ è una **funzione moltiplicativa**, ossia $\phi(a \cdot b) = \phi(a) \cdot \phi(b)$ per a, b **coprimi**

---

[2] o *primi tra loro*

[3] letto "*phi* di n"

Numeri coprimi

Phi di Eulero

Due interi a, b si dicono *coprimi*[2] se il loro massimo comun divisore (gcd) è 1.

- 1 è coprimo con tutti
- 0 non è coprimo con nessun intero ≥ 2

Sia $\phi(n)$[3] il numero di interi coprimi con n nel range $[1, n]$.

- $\phi(p) = p - 1$ per ogni p primo
- in generale, $\phi(p^k) = (p-1)p^{k-1} \ \forall k \in \mathbb{Z}_+$
- ϕ è una **funzione moltiplicativa**, ossia $\phi(a \cdot b) = \phi(a) \cdot \phi(b)$ per a, b **coprimi**

Ok ma perchè dovrebbe interessarci?

---

[2] o *primi tra loro*

[3] letto "*phi* di n"

Teorema di Eulero

Inversi modulari

Teorema di Eulero

Per ogni coppia di interi a, m coprimi,

$$a^{\phi(m)} \equiv 1 \pmod{m}.$$

Teorema di Eulero

Inversi modulari

Teorema di Eulero

Per ogni coppia di interi a, m coprimi,

$$a^{\phi(m)} \equiv 1 \pmod{m}.$$

In maniera equivalente,

$$a \cdot (a^{\phi(m)-1}) \equiv 1 \pmod{m}.$$

Chiamiamo **inverso modulare** di a un intero che moltiplicato con a è congruo a 1 $\pmod{m}$, $a^{\phi(m)-1}$ è un inverso modulare di a.

Teorema di Eulero

Inversi modulari

Teorema di Eulero

Per ogni coppia di interi a, m coprimi,

$$a^{\phi(m)} \equiv 1 \pmod{m}.$$

In maniera equivalente,

$$a \cdot (a^{\phi(m)-1}) \equiv 1 \pmod{m}.$$

Chiamiamo **inverso modulare** di a un intero che moltiplicato con a è congruo a 1 $\pmod{m}$, $a^{\phi(m)-1}$ è un inverso modulare di a.

Allora (se abbiamo un inverso) possiamo fare le divisioni!

Sotto modulo *mod* primo, invece che dividere per x moltiplichiamo per x^{p-2}.

Permutazioni

Permutazioni

Quante sono le permutazioni di n elementi?

[4] letto "n fattoriale"

Permutazioni

Permutazioni

Quante sono le permutazioni di n elementi?

Per la prima posizione abbiamo n possibilità, per la seconda $n-1$, per la terza $n-2$ e così via.

[4] letto "n fattoriale"

Permutazioni

Permutazioni

Quante sono le permutazioni di n elementi?

Per la prima posizione abbiamo n possibilità, per la seconda $n-1$, per la terza $n-2$ e così via.

La risposta è quindi $n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot 1$. Indichiamo questo prodotto come $n!$[4].

[4] letto "n fattoriale"

Disposizioni

Disposizioni

In quanti sono i possibili podi di una gara di n partecipanti?

Disposizioni

Disposizioni

In quanti sono i possibili podi di una gara di n partecipanti?

Per la prima posizione abbiamo n possibilità, per la seconda $n-1$, per la terza $n-2$.

Disposizioni

Disposizioni

In quanti sono i possibili podi di una gara di n partecipanti?

Per la prima posizione abbiamo n possibilità, per la seconda $n-1$, per la terza $n-2$.

La risposta è quindi $n \cdot (n-1) \cdot (n-2)$.

Disposizioni

Disposizioni

In quanti sono i possibili podi di una gara di n partecipanti?

Per la prima posizione abbiamo n possibilità, per la seconda $n-1$, per la terza $n-2$.

La risposta è quindi $n \cdot (n-1) \cdot (n-2)$.

In generale le disposizioni di n elementi in k posizioni sono

$$\frac{n!}{(n-k)!}$$

Combinazioni

Combinazioni

In quanti possiamo scegliere k elementi da un gruppo di n elementi? (non importa l'ordine)

Combinazioni

Combinazioni

In quanti possiamo scegliere k elementi da un gruppo di n elementi? (non importa l'ordine)

Il problema è simile al precedente, ma ora non importa l'ordine degli elementi. Ci basta quindi dividere per $k!$.

Combinazioni

Combinazioni

In quanti possiamo scegliere k elementi da un gruppo di n elementi? (non importa l'ordine)

Il problema è simile al precedente, ma ora non importa l'ordine degli elementi. Ci basta quindi dividere per $k!$. La risposta è

$$\frac{n!}{(n-k)! \cdot k!}$$

Combinazioni

Combinazioni

In quanti possiamo scegliere k elementi da un gruppo di n elementi? (non importa l'ordine)

Il problema è simile al precedente, ma ora non importa l'ordine degli elementi. Ci basta quindi dividere per $k!$. La risposta è

$$\frac{n!}{(n-k)! \cdot k!}$$

Questa formula viene detta **Binomiale** e si indica come

$$\binom{n}{k}$$

Combinazioni

Combinazioni

In quanti possiamo scegliere k elementi da un gruppo di n elementi? (non importa l'ordine)

Il problema è simile al precedente, ma ora non importa l'ordine degli elementi. Ci basta quindi dividere per $k!$. La risposta è

$$\frac{n!}{(n-k)! \cdot k!}$$

Questa formula viene detta **Binomiale** e si indica come

$$\binom{n}{k}$$

Nei problemi si chiede quasi sempre la risposta mod $10^9 + 7$. Di solito conviene precalcolare i fattoriali fino a $MAXN$ e i loro inversi modulari, per poi calcolare i binomiali all'occorrenza.

Implementazione

```cpp
static const ll mod = 1000000007;
vector<ll> fact = {1};
vector<ll> invfact = {1};

ll binom(int n, int k) {
    if (k < 0 || k > n) return 0;
    return fact[n] * invfact[k] % mod * invfact[n - k] % mod;
}

void precalc(int n) {
    for (int i = 1; i <= n; i++) {
        fact.push_back(fact.back() * i % mod);
        invfact.push_back(pot(fact[i], mod - 2) % mod);
    }
}
```

String Matching

String Matching

Date N stringhe, controlla se ci sono due stringhe uguali.

String Matching

String Matching

Date N stringhe, controlla se ci sono due stringhe uguali.

Risolvere il problema in modo naive richiederebbe $O(N^2)$, ma possiamo fare meglio.

String Matching

String Matching

Date N stringhe, controlla se ci sono due stringhe uguali.

Risolvere il problema in modo naive richiederebbe $O(N^2)$, ma possiamo fare meglio.

Un modo semplice è utilizzare una **unordered_set** o una **unordered_map** e inserire le stringhe una alla volta, controllando se la stringa che stiamo inserendo è già presente.

Ma esattamente come funzionano queste strutture dati?

Ma esattamente come funzionano queste strutture dati?

Quello che ci interessa è che convertono una stringa in un numero, utilizzando una **funzione di hash**. La funzione di hash deve essere **deterministica**, ovvero dato lo stesso input, deve restituire sempre lo stesso output.

Ma esattamente come funzionano queste strutture dati?

Quello che ci interessa è che convertono una stringa in un numero, utilizzando una **funzione di hash**. La funzione di hash deve essere **deterministica**, ovvero dato lo stesso input, deve restituire sempre lo stesso output.

In questo modo se due stringhe hanno hash diversi, allora sono sicuramente diverse, mentre se hanno hash uguali sono uguali **con una certa probabilità**.

Ma esattamente come funzionano queste strutture dati?

Quello che ci interessa è che convertono una stringa in un numero, utilizzando una **funzione di hash**. La funzione di hash deve essere **deterministica**, ovvero dato lo stesso input, deve restituire sempre lo stesso output.

In questo modo se due stringhe hanno hash diversi, allora sono sicuramente diverse, mentre se hanno hash uguali sono uguali **con una certa probabilità**.
(Per evitare collisioni in alcuni problemi è necessario utilizzare più funzioni di hash).

Vediamo ora un esempio di funzione di hash, detto Rabin-Karp, che possiamo sfruttare per risolvere problemi anche più complessi.

[5] in realtà è sufficiente siano primi tra loro

Vediamo ora un esempio di funzione di hash, detto Rabin-Karp, che possiamo sfruttare per risolvere problemi anche più complessi.

Algoritmo

Vediamo ogni stringa come un polinomio, ad esempio la stringa "abc" come "a" $\cdot x^2 +$ "b" $\cdot x +$ "c"
(dove "a" è il valore ascii del carattere 'a').

[5] in realtà è sufficiente siano primi tra loro

Vediamo ora un esempio di funzione di hash, detto Rabin-Karp, che possiamo sfruttare per risolvere problemi anche più complessi.

Algoritmo

Vediamo ogni stringa come un polinomio, ad esempio la stringa "abc" come "a" $\cdot x^2 +$ "b" $\cdot x +$ "c"
(dove "a" è il valore ascii del carattere 'a').
Scelti due interi p e m, calcoliamo il valore del polinomio in p modulo m, ovvero $(97 \cdot p^2 + 98 \cdot p + 99) \pmod{m}$.

Nota

- p e m si scelgono primi[5]
- m deve essere abbastana grande (ad esempio $10^9 + 7$ o $10^9 + 9$).
- p deve essere maggiore della dimensione dell'alfabeto (ad esempio un numero primo grande circa 200-300).

[5] in realtà è sufficiente siano primi tra loro

Implementazione

```
static const ll p = 241, m = 1000000007;
ll hash(string a) {
    ll ans = 0;
    for (int i = 0; i < a.size(); i++) {
        ans = (ans * p + a[i]) % m;
    }
    return ans;
}
```

Perché questo algoritmo è utile?

Perché questo algoritmo è utile?
Possiamo facilmente aggiornare l'hash di una stringa per calcolare quello di una stringa "simile", ad esempio:

Perché questo algoritmo è utile?
Possiamo facilmente aggiornare l'hash di una stringa per calcolare quello di una stringa "simile", ad esempio:

- $hash(s[a, b+1]) = hash(s[a, b]) \cdot p + s[b+1] \mod m$.
- $hash(s[a+1, b]) = (hash(s[a, b]) - s[a] \cdot p^{b-a}) \mod m$.
- $hash(s[k+1, b]) = hash(s[a, b]) - (hash(s[a, k]) \cdot p^{b-k}) \mod m$.

Perché questo algoritmo è utile?
Possiamo facilmente aggiornare l'hash di una stringa per calcolare quello di una stringa "simile", ad esempio:

- $hash(s[a, b+1]) = hash(s[a, b]) \cdot p + s[b+1] \mod m.$
- $hash(s[a+1, b]) = (hash(s[a, b]) - s[a] \cdot p^{b-a}) \mod m.$
- $hash(s[k+1, b]) = hash(s[a, b]) - (hash(s[a, k]) \cdot p^{b-k}) \mod m.$

Proprio per questo motivo questo algoritmo è anche chiamato **Rolling Hash**. Questo è molto utile per risolvere numerosi problemi su stringhe.

Pattern Matching

Date due stringhe s e t, conta il numero di occorrenze di t in s.
https://cses.fi/problemset/task/1753/

Pattern Matching

Date due stringhe s e t, conta il numero di occorrenze di t in s.
`https://cses.fi/problemset/task/1753/`

Esistono numerosi algoritmi per risolvere questo problema, come Z-Algorithm, KMP, etc...

Pattern Matching

Date due stringhe s e t, conta il numero di occorrenze di t in s.
`https://cses.fi/problemset/task/1753/`

Esistono numerosi algoritmi per risolvere questo problema, come Z-Algorithm, KMP, etc...
Possiamo facilmente risolverlo con Rolling Hash:

- Siano N la lunghezza di s e M la lunghezza di t.
- Calcoliamo l'hash di t e chaimiamolo $target$.
- Calcoliamo l'hash dei primi M caratteri di s e chiamiamolo $curr$.
- Aggiorniamo $curr$ aggiungiendo il carattere successivo di s e togliendo il primo.
- Ogni volta che $curr$ è uguale a $target$ abbiamo trovato un match (molto probabilmente).

Problemi

https://cses.fi/problemset/task/1095
https://cses.fi/problemset/task/1712
https://training.olinfo.it/#/task/ois_scount/statement
https://training.olinfo.it/#/task/ois_walker/statement
https://training.olinfo.it/#/task/ois_casino/statement
https://training.olinfo.it/#/task/itoi_morse/statement
https://training.olinfo.it/#/task/unimi_glitch/statement