Algorithme de Mo Tao et décomposition sqrt

Aymane Moutei

July 20, 2023

Outline

Introduction

La décomposition SQRT est une méthode qui permet d'effectuer certaines opérations (calcul de sommes, recherche min/max ou la médiane, etc...) en une complexité de $\mathcal{O}(Q * \sqrt{n})$, ce qui est beaucoup plus rapide que celle des algorithmes bruteforce.
Tout d'abord, nous décrirons la structure de données pour l'une des applications les plus simples de cette idée, puis nous montrerons comment la généraliser pour résoudre d'autres problèmes, et enfin nous examinerons une utilisation légèrement différente de cette idée : L'algorithme offline de de Mo Tao qui suit le meme paradigme.

Décomposition sqrt

Un premier problème statique

Problème 1

Etant donné un tableau t de taille $n \leq 10^5$ et un entier $q \leq 10^5$ représentant le nombre de requêtes à répondre telles que chaque requete contient 2 entiers l et r. La réponse à chacune doit être la somme $\sum_{i=l}^{r} t[i]$

Décomposition sqrt

Un premier problème statique

Approche naive:

Bruteforce, complexité $\mathcal{O}(qn)$

Approche efficace:

On précalcule la liste préfixe c'est-à-dire $pref[k] = \sum_{i=0}^{k} t[i]$ en $\mathcal{O}(n)$ et la réponse à chaque requete peut être calculée en $\mathcal{O}(1)$ car $\sum_{i=l}^{r} t[i] = pref[r] - pref[l-1]$ pour $l > 0$ et $\sum_{i=l}^{r} t[i] = pref[r]$ pour $l = 0$.

Cela donne une complexité totale de $\mathcal{O}(n+q)$ qui est également la meilleure que l'on puisse obtenir.

Décomposition sqrt

Un deuxiéme problème statique

Problème 2

étant donné un tableau t de taille $n \leq 10^5$ et un entier $q \leq 10^5$ représentant le nombre de requêtes à répondre telles que chaque requete contenant 2 entiers l et r. On définit le diamètre d'un segment $[l; r]$ comme étant égal à

$$\max_{i \in [l;r]} t[i] - \min_{i \in [l;r]} t[i]$$

La réponse à chaque requete doit être le diamètre de $[l; r]$.

Décomposition sqrt

Un deuxiéme problème statique

Approche efficace:

La meilleure approche consiste à utiliser une structure de données appelée "Sparse Table". On précalcule $mx[i][k] = \max_{j \in [i; i+2^k-1]} t[i]$ et $mn[i][k] = \min_{j \in [i; i+2^k-1]} t[i]$ en $\mathcal{O}(n * log(n))$, en éxploitant la relation de réccurence $mx[i][k] = max(mx[i][k-1], mx[i+2^{k-1}][k-1])$. Chaque requête peut être traitée en $\mathcal{O}(log(n))$ en décomposant $r - l$ en base 2 et en faisant des sauts de puissances de 2, ce qui fait une complexité totale de $\mathcal{O}((n+Q) * log(n))$

```cpp
#include<bits/stdc++.h>
using namespace std;
int main(){
    int n;
    cin>>n;
    int k = log2(n)+2;
    int t[n],sp[n][k],spp[n][k];
    for(int i=0;i<n;i++){
        cin>>t[i];
        sp[i][0]= t[i];
        spp[i][0]= t[i];
    }
    for(int j=1;j<n;j++)
        for(int i=0;i+(1<<j)<=n;i++){
            sp[i][j] = max(sp[i][j-1],sp[i+(1<<(j-1))][j-1])
    ;
            spp[i][j] = min(spp[i][j-1],spp[i+(1<<(j-1))][j
    -1]);
        }
    int q;
    cin>>q;
    while(q--){
        int l,r;
```

```
        cin>>l>>r;
        l--,r--;
        int m = r-l+1;
        int mxx = -1e9,mnn= 1e9;
        for(int i=0;i<30;i++)
            if((1<<i)&m){
                mxx=max(mxx,sp[l][i]);
                mnn=min(mnn,spp[l][i]);
                l+=(1<<i);
            }
        cout<<mxx-mnn<<endl;
    }
}
```

Listing 1: Implémentation de Sparse Table pour résoudre le problème du diamètre statique

Décomposition sqrt

Où la décomposition sqrt semble utile

Problème 3

Étant donné un tableau t de taille $n \leq 10^5$ et un entier $q \leq 10^5$ représentant le nombre de requêtes. Les requêtes sont de 2 types:

1. on modifie l'élement i ème, c'est-à-dire $t[i] := x$.
2. on affiche le diamètre de $[l; r]$

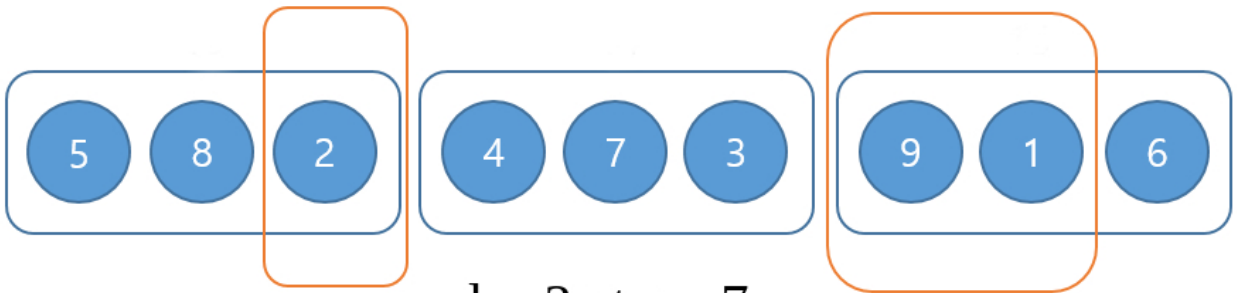
La solution la plus efficace a une complexité de $\mathcal{O}((n + Q) * log(n))$ en utilisant une structure de donnée connue sous le nom de "Segment Tree". Nous proposons une approche basée sur la décomposition sqrt, qui a l'avantage d'être rapide à implémenter.

Décomposition sqrt

On commence par découper notre liste en des blocs chacun de taille d (On verra par la suite que la complexité temporelle est optimale pour $d = \sqrt{n}$ d'où le nom) Nous divisons donc le problème en traitant chaque bloc individuellement.

Nous notons, pour tout $i \in [0; \lfloor n/d \rfloor]$: $mx[i] = \max_{j \in [i*d; i*(d+1)-1]} t[i]$ et de meme $mn[i] = \min_{j \in [i*d; i*(d+1)-1]} t[i]$.

Pour répondre à ue requete du 2ème type, on décompose notre intevalle $[l; r]$ en des blocs de taille d totalement inclus dans $[l; r]$dont on connait dèja les élements maximals et minimals. Il nous reste eventuellement une queue et/ou une tete de taille inférieure ou égale à d. On peut donc calculer l'élement maximal et minimal de $[l; r]$ en $\mathcal{O}(d + n/d)$.



```cpp
#include<bits/stdc++.h>
using namespace std;
int main(){
    int n,q,d;
    cin>>n>>q;
    d = sqrt(n);
    vector<int> t(n),mxx(d,-1e9),mnn(d,1e9);
    for(auto &x:t)
        cin>>x;
    for(int i=0;i<d;i++)
        for(int j=0;j<d;j++){
            mxx[i]=max(mxx[i],t[i+j*d]);
            mnn[i]=min(mnn[i],t[i+j*d]);
        }
    while(q--){
        int dum;
        cin>>dum;
        if(dum==1){
            int id,x;
            cin>>id>>x;
            int i = id/d;
            t[id]=x,mxx[i]=-1e9,mnn[i]=1e9;
```

```
24              for(int j=0;j<d;j++){
25                  mxx[i]=max(mxx[i],t[i+j*d]);
26                  mnn[i]=min(mnn[i],t[i+j*d]);
27              }
28          }
29          else{
30              int l,r;
31              cin>>l>>r;
32              int ll=ceil(double(l)/d),rr=r/d-1;
33              int mx=-1e9,mn=1e9;
34              if(ll<=rr){
35                  for(int i=ll;i<=rr;i++){
36                      mx=max(mx,mxx[i]);
37                      mn=min(mn,mnn[i]);
38                  }
39                  for(int i=l;i<ll*d;i++){
40                      mx=max(mx,t[i]);
41                      mn=min(mn,t[i]);
42                  }
43                  for(int i=d*rr;i<=r;i++){
44                      mx=max(mx,t[i]);
45                      mn=min(mn,t[i]);
46                  }
```

```
47                  cout<<mx-mn<<endl;
48              }
49              else{
50                  for(int i=l;i<=r;i++){
51                      mx=max(mx,t[i]);
52                      mn=min(mn,t[i]);
53                  }
54                  cout<<mx-mn<<endl;
55              }
56          }
57      }
58  }
```

Listing 2: Implémentation C++ du problème du diamètre

Décomposition sqrt

Un Problème plus avancé

Problème 4

Il y a n trampoline disposés sur une seule rangée et numérotés de gauche à droite avec des numéros de 1 à n. Chaque trampoline a sa propre puissance (la i ème a une puissance t_i). Si vous lancez une balle dans la position i, elle sautera immédiatement vers la position $i + t_i$ et ainsi de suite jusqu'à sortir complétement de la rangée. Lors de chacunes des Q requêtes données, On doit effectuer l'une des deux actions suivantes :

1. Changer la puissance de la i ème trampoline à p cad: $t[i] := p$.
2. Lancer une balle à la position i et afficher le nombre de sauts que la balle a effectué avant qu'elle ne sorte de la rangée, et également la dérnière position à partir de laquelle elle a sauté juste avant de quitter la rangée.

Décomposition sqrt

problème de trampoline

Approche efficace:

Comme vu précedement, on découpe notre liste en des blocs chacun de taille d ($d = \sqrt{n}$ pour un temps d'éxecution optimal).
Pour tout i dans [1;n] on note:
$depth[i]$ = le nombre de sauts efféctués avant de sortir du block associé à i
$in[i]$ = l'indice de la dérnière position ou l'on quitte le block actuel.
$out[i]$ = l'indice de la première position après avoir quitté le bloc associé à i

Remarque 1: On peut précalculer ces deux tableaux pour chaque bloc en faisant un scan linéaire de la droite vers la gauche en distinguant les cas ou un saut va nous faire sortir du block actuel et le cas opposé. Dans ce cas on a les relations de réccurence suivantes:

$$\begin{cases} depth[i] &= depth[i + t[i]] + 1 \\ in[i] &= in[i + t[i]] \\ out[i] &= out[i + t[i]] \end{cases}$$

avec les cas de base suivants :

$$\begin{cases} depth[i] &= 0 \\ in[i] &= i \\ out[i] &= i + t[i] \end{cases}$$

Remarque 2: La complexité de la requete de modification est de $\mathcal{O}(d)$, en effet les tableaux *in*,*out* et *depth* ne sont modifiés que pour les valeurs i dans le block, la modification se faisant par un scan linéaire de la droite vers la gauche d'ou la complexite $\mathcal{O}(d)$

Remarque 3: En ce qui concerne les requêtes ou l'on lançe la balle à l'indice i, on parcourt chaque bloc dans $\mathcal{O}(1)$ puisqu'on sait dèja les valeurs du tableau *out* qui indique le premier indice ou l'on sort du block, il suffit donc d'iterer *out* tout en sommant les *depth* rencontrés le long du chemin. Comme il y a n/d blocs la complexité d'une telle requete est $\mathcal{O}(n/d)$

On conclut par les 3 remarques que le problème se résout par la paradigme de "sqrt décomposition" en une complexité totale moyenne de $\mathcal{O}(q * (n + n/d))$ qui est minimale pour $d = \sqrt{n}$ d'ou la complexité finale de $\mathcal{O}(q * \sqrt{n})$

```cpp
#include<bits/stdc++.h>
#pragma GCC target ("avx2")
#pragma GCC optimization ("O3")
#pragma GCC optimization ("unroll-loops")
#define F first
#define S second
#define PB push_back
#define MP make_pair
#define all(c) c.begin(), c.end()
using namespace std;
vector<int> t,in,out,depth;
int n,m,r,type,a,b;
void upd(int l){
    for(int i=min(l+r-1,n-1);i>=l;i--)
        if(i+t[i]<=min(l+r-1,n-1))
            in[i]=in[i+t[i]],depth[i]=depth[i+t[i]]+1;
        else{
            in[i]=i;
            depth[i]=0;
            if(i+t[i]<=n-1)
                out[i]=i+t[i];
            else
                out[i]=-1;
```

```
24          }
25  }
26  int main(){
27      cin>>n>>m;
28      r=sqrt(n);
29      t=vector<int>(n),in=vector<int>(n),out=vector<int>(n),
        depth=vector<int>(n);
30      for(auto &x:t)  scanf("%d",&x);
31      for(int i=0;i<=r+1;i++)
32          if(i*r<n)
33              upd(i*r);
34      while(m--){
35          scanf("%d%d",&type,&a);
36          if(type==0){
37              scanf("%d",&b);
38              a--;
39              t[a]=b;
40              upd((a/r)*r);
41          }
42          else{
43              a--;
44              int d=0;
45              while(1){
```

```
46                  if(in[a]==a){
47                      if(out[a]==-1)
48                          break;
49                      d++;
50                      a=out[a];
51                  }
52                  else{
53                      d+=depth[a];
54                      a=in[a];
55                  }
56              }
57              printf("%d %d\n",a+1,d+1);
58          }
59      }
60  }
```

Listing 3: Implémentation C++ du problème de trampoline

Algorithme de Mo

utilisé par un contestant aux Olympiades d'informatique chinoises en 2009, L'algorithme de Mo (en hommage à ce soldat inconnu) a été popularisé dans la communauté de Programmation compétitive. Il permet de résoudre les requêtes de manière "offline" cad la réponse à chaque requete n'est pas fournit sur place mais plutot les requêtes sont réordronnées de telle manière à minimiser le temps d'éxecution. Pour mieux comprendre, Considérons le problème suivant:

Problème

On dispose d'une liste d'entiers t de taille n et de Q requêtes chacune demandant d'afficher l'élement le plus fréquent dans l'intervalle $[l; r]$.

Bruteforce: l'approche naive auquelle on peut penser au début consiste à itérer sur l'intervalle à chaque requete et en utilisant une map on obtient façilement l'élement le plus réccurent. Ce qui fait une complexité de $\mathcal{O}(q * n)$.
2 ème essaie: L'algorithme de Mo peut de découper en 2 étapes :
On commence par ordonner les requêtes (l_i, r_i) selon la relation d'ordre suivante:

$$(l_i, r_i) \preceq (l_j, r_j) \Longleftrightarrow \begin{cases} l_i/\sqrt{n} < l_j/\sqrt{n} \\ (l_i < l_j \text{ ou } (l_i = l_j \text{ et } r_i \leq r_j) \text{ si } l_i/\sqrt{n} = l_j/\sqrt{n} \end{cases}$$

On procède après les requêtes une par une en utilisant 2 variables l et r qui reprensentent la fenetre ou l'on est actuellement.à chaque fois on étend ou restreint la fenetere jusqu'à couvrir exactement l'intervalle de la requete actuelle.

```
for(auto &x:qe){
    while(curr<x.r)
        add(++curr);
    while(curl>x.l)
        add(--curl);
    while(curr>x.r)
        rem(curr--);
    while(curl<x.l)
        rem(curl++);
    ans[x.idx]=extract();
    //show();
}
```

Listing 4: Impémentation des fonctions qui adapte l'intervalle actuel en C++

On définit les fonctions qui permettent d'ajouter un élement ou l'enlever de notre set qui ne garde que les élements (et leurs occurences) de la fenetre $[cur_l, cur_r]$

```
void add(int i){
    auto it = in.find(MP(occ[t[i]],t[i]));
    if(it != in.end())
```

```
        in.erase(it);
    occ[t[i]]+=1;
    in.insert(MP(occ[t[i]],t[i]));
}
void rem(int i){
    auto it = in.find(MP(occ[t[i]],t[i]));
    if(it != in.end())
        in.erase(it);
    occ[t[i]]-=1;
    in.insert(MP(occ[t[i]],t[i]));
}
```

Listing 5: Implémentation des fonctions d'ajout/suppression en C++

```
#include<bits/stdc++.h>
#define MP make_pair
using namespace std;
int blck;
struct Query{
    int idx,l,r;
};
set<pair<int,int>,greater<pair<int,int>> > in;
map<int,int> occ;
vector<int> t;
```

```
vector<pair<int,int>> ans;
vector<Query> qe;
bool cmp(Query a,Query b){
    if(a.l/blck != b.l/blck)
        return a.l/blck < b.l/blck;
    if(a.l != b.l)
        return a.l < b.l;
    return a.r < b.r;
}
void add(int i){
    auto it = in.find(MP(occ[t[i]],t[i]));
    if(it != in.end())
        in.erase(it);
    occ[t[i]]+=1;
    in.insert(MP(occ[t[i]],t[i]));
}
void rem(int i){
    auto it = in.find(MP(occ[t[i]],t[i]));
    if(it != in.end())
        in.erase(it);
    occ[t[i]]-=1;
    in.insert(MP(occ[t[i]],t[i]));
}
```

```
34 pair<int,int> extract(){
35     auto x = *in.begin();
36     return MP(x.second,x.first);
37 }
38 void show(){
39     for(auto it = in.begin();it != in.end();++it){
40         auto x = *it;
41         cout<<x.first<<" "<<x.second<<endl;
42     }
43     cout<<endl;
44 }
45 int main(){
46     int n,q;
47     cin>>n>>q;
48     t = vector<int> (n);
49     ans = vector<pair<int,int>> (q);
50     blck = sqrt(n);
51     for(auto &x:t)
52         cin>>x;
53     for(int i=0;i<q;i++){
54         int l,r;
55         cin>>l>>r;
56         Query temp;
```

```
57          temp.idx=i,temp.l=l,temp.r=r;
58          qe.push_back(temp);
59      }
60      sort(qe.begin(),qe.end(),cmp);
61      int curl=0,curr=-1;
62      for(auto &x:qe){
63          while(curr<x.r)
64              add(++curr);
65          while(curl>x.l)
66              add(--curl);
67          while(curr>x.r)
68              rem(curr--);
69          while(curl<x.l)
70              rem(curl++);
71          ans[x.idx]=extract();
72          //show();
73      }
74
75      for(auto x:ans)
76          cout<<x.first<<" "<<x.second<<endl;
77
```

```
78 }
```

Listing 6: Implémentation de l'algorithme de Mo en C++

Complexité?

Le Tri des requêtes se fait en $\mathcal{O}(Q * log(Q))$.
Qu'en est-il des autres opérations ? Combien de fois les fonctions *add* et *rem* sont-elles appelées?
Supposons que la taille des blocs soit d.
Si nous ne considérons que les requêtes ayant l'indice gauche l dans le même bloc, les requêtes sont triées par l'indice droit r. Par conséquent, nous appellerons $add(curr)$ et $remove(curr)$ seulement $\mathcal{O}(n)$ fois pour l'ensemble de ces requêtes.
Cela fait donc $\mathcal{O}(\frac{n}{d} * n)$ appels pour toutes les requêtes des différents blocs.
La valeur de *curl* peut changer d'au plus $\mathcal{O}(d)$ entre deux requêtes du meme bloc. Par conséquent, nous avons $\mathcal{O}(d * Q)$ appels supplémentaires de $add(curl)$ et $remove(curl)$.
On a donc un total de $\mathcal{O}(\frac{n}{d} * n + d * Q)$ opérations.

Complexité?

En particulier, Pour $d \approx \sqrt{n}$, cela donne un total de $\mathcal{O}((n+Q)\sqrt{n})$ opérations. Ainsi, la complexité est de $\mathcal{O}((n+Q)F\sqrt{n})$ où $\mathcal{O}(F)$ est la complexité des fonctions *add* et *rem*.
Dans notre cas, on a implémenté ces fonctions de tel sorte à ce que leur complexité soit $\mathcal{O}(log(n))$.
La complexité finale est donc de $\mathcal{O}((n+Q) * log(n) * \sqrt{n})$

Annexes

La strcutre de donée "Segment tree" est très populaire dans le monde de la programmation compétitive. Ca nous permet de résoudre les problèmes du type "RURQ" càd à chaque requete soit on modifie le tableau qui contient nos élements soit on répond à la requete $f(a_l, ..., a_r)$ pour certains $l \leq r$ à condition que f soit associative.

Annexes

Problème d'application

Etant donné un entier $n \leq 10^5$ et une liste d'entiers $t_0, .., t_{n-1} \leq 10^9$ et un entier $Q \leq$ qui désigne le nombre de requetes. Chaque requete contient 2 entier $l \leq r$ et la réponse doit etre $\#\{i \in [l; r] | pgcd(t_l, .., t_r) = t_i\}$

```cpp
#include<bits/stdc++.h>
#pragma GCC optimize("O2")
#define F first
#define S second
#define PB push_back
#define MP make_pair
#define all(c) c.begin(), c.end()
#define endl "\n"
#define sz(u) (int)(u.size())
#define L(x)(2*x)
#define R(x)(2*x+1)
#define M(x,y)((x+y)/2)
#define int long long
typedef long long ll;
typedef unsigned long long ull;
using namespace std;
const int N=1e5+1;
vector<int> tree(4*N);
void update(int p,int l,int r,int pos,int val){
    if(l>pos || r<pos)
        return;
    if(l==r){
        tree[p]=val;
```

```
24          return;
25      }
26      update(L(p),l,M(l,r),pos,val);
27      update(R(p),M(l,r)+1,r,pos,val);
28      tree[p]=__gcd(tree[L(p)],tree[R(p)]);
29  }
30  int query(int p,int l,int r,int i,int j){
31      if(r<i || l>j)
32          return 0;
33      if(l>=i && r<=j)
34          return tree[p];
35      return __gcd(query(L(p),l,M(l,r),i,j),query(R(p),M(l,r)
        +1,r,i,j));
36  }
37  signed main(){
38      ios::sync_with_stdio(0);
39      cin.tie(0);
40      cout.tie(0);
41      int n;
42      cin>>n;
43      vector<int> t(n+1);
44      vector<pair<int,int>> st;
45      for(int i=1;i<=n;i++){
```

```
46          cin>>t[i];
47          update(1,1,N,i,t[i]);
48          st.PB({t[i],i});
49      }
50      sort(all(st));
51      int q;
52      cin>>q;
53      while(q--){
54          int l,r;
55          cin>>l>>r;
56          int gcd=query(1,1,N,l,r);
57          int num=upper_bound(all(st),MP(gcd,r))-lower_bound(
    all(st),MP(gcd,l));
58          cout<<num<<endl;
59      }
60
61  }
```

Listing 7: Implémentation de la solution proposée

References


Steven Halim.
Competitive Programming 4: The Lower Bound of Programming Contests in the 2020s.


Antti Laaksonen.
Competitive Programmer's Handbook.
https://cses.fi/book/book.pdf.


Ivanov Maxim.
https://cp-algorithms.com/.


Codeforces cp blogs.
https://codeforces.com/blog/entry/83248.


Codeforces problemset.
https://codeforces.com/problemset/problem/13/E.