

# Algorithme de Mo Tao et décomposition sqrt

Ayman Moutei

July 20, 2023

## Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Décomposition sqrt</b>	<b>1</b>
2.1	Un premier problème statique . . . . .	1
2.2	Où la décomposition sqrt semble utile . . . . .	2
2.3	Un Problème plus avancé . . . . .	3
<b>3</b>	<b>Algorithme de Mo</b>	<b>4</b>
3.1	Complexité? . . . . .	5
<b>4</b>	<b>Annexe</b>	<b>5</b>

## 1 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 Mo Tao qui suit le meme paradigme.

## 2 Décomposition sqrt

### 2.1 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  $\sum_{i=l}^r t[i]$

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.

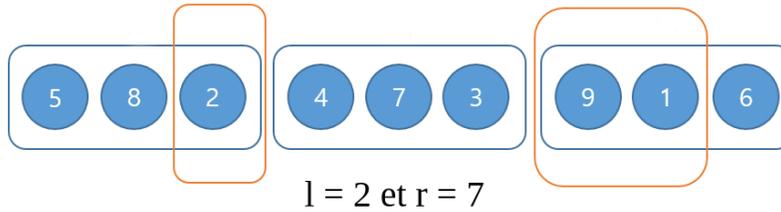


Figure 1: Exemple d'un découpage  $n = 9$  et  $d = 3$

### 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  $[l; r]$ .

#### Approche efficace:

La meilleure approche consiste à utiliser une structure de données appelée "Sparse Table". On précalcule  $mx[i][k] = \max_{j \in [i; i+2^k-1]} t[j]$  et  $mn[i][k] = \min_{j \in [i; i+2^k-1]} t[j]$  en  $\mathcal{O}(n * \log(n))$ , en exploitant la relation de récurrence  $mx[i][k] = \max(mx[i][k-1], mx[i+2^{k-1}][k-1])$ . 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  $\mathcal{O}((n+Q) * \log(n))$ . Voir annexe pour l'implémentation.

## 2.2 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'élément  $i$  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.

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

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 connaît déjà les éléments maximaux et minimaux. Il nous reste éventuellement une queue et/ou une tete de taille inférieure ou égale à  $d$ . On peut donc calculer l'élément maximal et minimal de  $[l; r]$  en  $\mathcal{O}(d + n/d)$ .

Quant aux requêtes du premier type la modification du  $i$  ème élément ne concerne que le  $\lfloor i/d \rfloor$  bloc, il suffit donc de recalculer  $mn[\lfloor i/d \rfloor]$  et  $mx[\lfloor i/d \rfloor]$  chose qu'on peut faire en  $\mathcal{O}(d)$  On remarque que la complexité moyenne d'une requete est minimale pour  $d = \sqrt{n}$  ce qui fait une complexité finale de  $\mathcal{O}(Q * \sqrt{n})$

Voir annexe pour l'implémentation

## 2.3 Un Problème plus avancé

Voilà un problème plus subtile qui nécessite l'utilisation du paradigme de décomposition sqrt pour réduire la complexité de l'approche naive qui est  $\mathcal{O}(n * Q)$  à  $\mathcal{O}(\sqrt{n} * Q)$

### Problème 4

Il y a  $n$  trampolines 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 complètement de la rangée. Lors de chacune 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 dernière position à partir de laquelle elle a sauté juste avant de quitter la rangée.

Approche efficace:

Comme vu précédemment, on découpe notre liste en des blocs chacun de taille  $d$  ( $d = \sqrt{n}$  pour un temps d'exécution optimal).

Pour tout  $i$  dans  $[1;n]$  on note  $depth[i]$  = le nombre de sauts effectués avant de sortir du block associé à  $i$  et  $in[i]$  = l'indice de la dernière position ou l'on quitte le block actuel. et  $out[i]$  = 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 récurrence 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'où la complexité  $\mathcal{O}(d)$

Remarque 3: En ce qui concerne les requêtes ou l'on lance la balle à l'indice  $i$ , on parcourt chaque bloc dans  $\mathcal{O}(1)$  puisqu'on sait déjà 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'où la complexité finale de  $\mathcal{O}(q * \sqrt{n})$   
 Voir annexe pour l'implémentation

### 3 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 réordonnées de telle manière à minimiser le temps d'exécution. 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'élément le plus fréquent dans l'intervalle  $[l; r]$ .

Bruteforce: l'approche naive auquel on peut penser au début consiste à itérer sur l'intervalle à chaque requete et en utilisant une map on obtient facilement l'élément 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:

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

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

```

1
2 for(auto &x:qe){
3     while(curr<x.r)
4         add(++curr);
5     while(curl>x.l)
6         add(--curl);
7     while(curr>x.r)
8         rem(curr--);
9     while(curl<x.l)
10        rem(curl++);
11    ans[x.idx]=extract();
12    //show();
13 }
```

Listing 1: Impémentation des fonctions qui adapte l'intervalle actuel en C++

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

```

1 void add(int i){
2     auto it = in.find(MP(occ[t[i]],t[i]));
3     if(it != in.end())
4         in.erase(it);
5     occ[t[i]]+=1;
6     in.insert(MP(occ[t[i]],t[i]));
7 }
8 void rem(int i){
9     auto it = in.find(MP(occ[t[i]],t[i]));
10    if(it != in.end())
11        in.erase(it);
12    occ[t[i]]-=1;
13    in.insert(MP(occ[t[i]],t[i]));
14 }
```

Listing 2: Implémentation des fonctions d'ajout/suppression en C++

Le code final en annexe.

### 3.1 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 même 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.

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 telle sorte à ce que leur complexité soit  $\mathcal{O}(\log(n))$ .

La complexité finale est donc de  $\mathcal{O}((n + Q) * \log(n) * \sqrt{n})$

## 4 Annexe

```
1 #include <bits/stdc++.h>
2 using namespace std;
3 int main(){
4     int n;
5     cin >> n;
6     int k = log2(n)+2;
7     int t[n], sp[n][k];
8     for(int i=0; i<n; i++){
9         cin >> t[i];
10        sp[i][0] = t[i];
11    }
12    for(int j=1; j<n; j++){
13        for(int i=0; i+(1<<j) <= n; i++){
14            sp[i][j] = max(sp[i][j-1], sp[i+(1<<(j-1))][j-1]);
15        }
16    }
17    int q;
18    cin >> q;
19    while(q--){
20        int l, r;
21        cin >> l >> r;
22        l--, r--;
23        int m = r-l+1;
24        int ans = -1e9;
25        for(int i=0; i<30; i++){
26            if((1<<i) & m){
27                ans = max(ans, sp[l][i]);
28                l += (1<<i);
29            }
30        }
31        cout << ans << endl;
32    }
33 }
```

Listing 3: Implémentation de Sparse Table pour résoudre le problème de max range statique

```
1 #include <bits/stdc++.h>
2 using namespace std;
3 int main(){
4     int n, q, d;
5     cin >> n >> q;
6     d = sqrt(n);
```

```

8   vector<int> t(n),sum(d,0);
9   for(auto &x:t)
10      cin>>x;
11   for(int i=0;i<d;i++)
12      for(int j=0;j<d;j++)
13         sum[i]+=t[i+j*d];
14   while(q--){
15      int dum;
16      cin>>dum;
17      if(dum==1){
18         int id,x;
19         cin>>id>>x;
20         sum[id/d]-=t[id];
21         t[id]=x;
22         sum[id/d]+=x;
23      }
24      else{
25         int l,r;
26         cin>>l>>r;
27         int ll=ceil(double(l)/d),rr=r/d-1;
28         int ans=0;
29         if(ll<=rr){
30            for(int i=ll;i<=rr;i++)
31               ans+=sum[i];
32            for(int i=1;i<ll*d;i++)
33               ans+=t[i];
34            for(int i=d*rr;i<=r;i++)
35               ans+=t[i];
36            cout<<ans<<endl;
37         }
38         else{
39            for(int i=1;i<=r;i++)
40               ans+=t[i];
41            cout<<ans<<endl;
42         }
43      }
44   }
45 }

```

Listing 4: Implémentation C++ du problème du diamètre

```

1 #include <bits/stdc++.h>
2 #pragma GCC target ("avx2")
3 #pragma GCC optimization ("O3")
4 #pragma GCC optimization ("unroll-loops")
5 #define F first
6 #define S second
7 #define PB push_back
8 #define MP make_pair
9 #define all(c) c.begin(), c.end()
10 using namespace std;
11 vector<int> t,in,out,depth;
12 int n,m,r,type,a,b;
13 void upd(int l){
14     for(int i=min(l+r-1,n-1);i>=l;i--)
15         if(i+t[i]<=min(l+r-1,n-1))
16             in[i]=in[i+t[i]],depth[i]=depth[i+t[i]]+1;
17     else{
18         in[i]=i;
19         depth[i]=0;
20         if(i+t[i]<=n-1)
21             out[i]=i+t[i];
22         else
23             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 5: Implémentation C++ du problème de trampoline

```

1  #include<bits/stdc++.h>
2  #define MP make_pair
3  using namespace std;
4  int blk;
5  struct Query{
6      int idx,l,r;
7  };
8  set<pair<int,int>,greater<pair<int,int>> > in;
9  map<int,int> occ;
10 vector<int> t;
11 vector<pair<int,int>> ans;
12 vector<Query> qe;
13 bool cmp(Query a,Query b){
14     if(a.l/blk != b.l/blk)
15         return a.l/blk < b.l/blk;
16     if(a.l != b.l)
17         return a.l < b.l;
18     return a.r < b.r;
19 }
20 void add(int i){
21     auto it = in.find(MP(occ[t[i]],t[i]));
22     if(it != in.end())
23         in.erase(it);
24     occ[t[i]]+=1;
25     in.insert(MP(occ[t[i]],t[i]));
26 }
27 void rem(int i){
28     auto it = in.find(MP(occ[t[i]],t[i]));
29     if(it != in.end())
30         in.erase(it);
31     occ[t[i]]-=1;
32     in.insert(MP(occ[t[i]],t[i]));
33 }
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     blk = 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++

## References

- [1] Steven Halim. *Competitive Programming 4: The Lower Bound of Programming Contests in the 2020s*.
- [2] Antti Laaksonen. *Competitive Programmer's Handbook*. <https://cses.fi/book/book.pdf>.
- [3] Ivanov Maxim. <https://cp-algorithms.com/>.
- [4] Codeforces cp blogs. <https://codeforces.com/blog/entry/83248>.
- [5] Codeforces problemset. <https://codeforces.com/problemset/problem/13/E>.