Segment tree à propagation paresseuse générique

Hugo Peyraud-Magnin

France-IOI

Toussaint 2023

Surcharge de l'addition

En C++, les variables ont un **type** et les opérateurs (addition $+$, comparaison $<$, etc.) entre deux variables dépendent de leur type.

```
int a = 120, b = 23;
cout << (a + b) << '\n'; // 143
cout << (a < b) << '\n'; // false

string x = "120", y = "23";
cout << (x + y) << '\n'; // 12023
cout << (x < y) << '\n'; // true
```

Structures de données et algorithmes génériques

Il est alors possible de concevoir des structures de données qui s'adaptent à n'importe quel type. Par exemple `vector<T>` est un tableau d'éléments de type `T` et le tri `sort` dépend de l'opérateur $<$ du type `T`.

```
vector<int> v1 {120, 23};
sort(v1.begin(), v1.end());
cout << v1[0] << ", " << v1[1] << '\n'; // 23, 120

vector<string> v2 {"120", "23"};
sort(v2.begin(), v2.end());
cout << v2[0] << ", " << v2[1] << '\n'; // 120, 23
```

Notre but va être de concevoir un segment tree qui va contenir des éléments d'un certain type T (information sur intervalle) et permettre de calculer des « sommes » sur intervalle selon un opérateur $\oplus$ personnalisé.

```
int opMin(int x, int y) {
    return min(x, y);
}
int opSomme(int x, int y) {
    return x+y;
}
segtree<int, opMin> A({5, 9, 3, 7, 1});
cout << A.requete(1, 4) << '\n'; // 3
segtree<int, opSomme> B({5, 9, 3, 7, 1});
cout << B.requete(1, 4) << '\n'; // 19
```

Type personnalisé

On peut écrire une structure (ou classe) pour stocker plus d'informations sur les intervalles.

$$\underbrace{(a_l \oplus \cdots \oplus a_{m-1})}_{\text{x.min, x.somme}} \oplus \underbrace{(a_m \oplus \cdots \oplus a_r)}_{\text{y.min, y.somme}} = \underbrace{(a_l \oplus \cdots \oplus a_r)}_{\text{res.min, res.somme}}$$

```
struct node {
    int min, somme;
};
node op(node x, node y) {
    node res;
    res.min = min(x.min, y.min);
    res.somme = x.somme + y.somme;
    return res;
}
segtree<node, op> A(/* ... */);
```

Type personnalisé

On peut écrire une structure (ou classe) pour stocker plus d'informations sur les intervalles.

$$\underbrace{(a_l \oplus \cdots \oplus a_{m-1})}_{\text{x.min, x.somme}} \oplus \underbrace{(a_m \oplus \cdots \oplus a_r)}_{\text{y.min, y.somme}} = \underbrace{(a_l \oplus \cdots \oplus a_r)}_{\text{res.min, res.somme}}$$

```
struct node {
    int min, somme;
};
node op(node x, node y) {
    return {min(x.min, y.min), x.somme + y.somme};
}
segtree<node, op> A(/* ... */);
```

Somme des $i \cdot a_i$

Même lorsque l'on souhaite calculer un seul entier, il peut être nécessaire de stocker des **informations intermédiaires**. Par exemple, comment calculer rapidement

$$\mathrm{sIndVal}(l, r) = a_l + 2a_{l+1} + 3a_{l+2} + \cdots + (r - l)a_{r-1}$$

$$a_4 + 2a_5 + 3a_6 + 4a_7 \neq (a_4 + 2a_5) + (a_6 + 2a_7)$$

```
struct node {
    int sumIndVal, ???;
};
node op(node x, node y) {
    return { ??? };
}
```

Somme des $i \cdot a_i$

$$\mathrm{sIndVal}(l, r) = a_l + 2a_{l+1} + 3a_{l+2} + \cdots + (r-l)a_{r-1}$$

$$\begin{aligned} a_4 + 2a_5 + 3a_6 + 4a_7 = (a_4 + 2a_5) + (a_6 + 2a_7) \\ + 2(a_6 + a_7) \end{aligned}$$

```
struct node {
    int sumIndVal, ???;
};
node op(node x, node y) {
    return { ??? };
}
```

$$\mathrm{sIndVal}(l, r) = a_l + 2a_{l+1} + 3a_{l+2} + \cdots + (r - l)a_{r-1}$$

	$1a_4$	$2a_5$	$3a_6$	$4a_7$	$1a_8$	$2a_9$	$3a_{10}$	$4a_{11}$
$+$					$4a_8$	$4a_9$	$4a_{10}$	$4a_{11}$
$=$	$1a_4$	$2a_5$	$3a_6$	$4a_7$	$5a_8$	$6a_9$	$7a_{10}$	$8a_{11}$

```
struct node {
    int sumIndVal, ???;
};
node op(node x, node y) {
    return { ??? };
}
```

Somme des $i \cdot a_i$

$$\mathrm{sIndVal}(l, r) = a_l + 2a_{l+1} + 3a_{l+2} + \cdots + (r-l)a_{r-1}$$

$$\sum_{l \to r}(i - l + 1)a_i = \sum_{l \to m}(i - l + 1)a_i + \sum_{m \to r}(i - m + 1)a_i + (m - l)\sum_{m \to r} a_i$$

```
struct node {
    int sumIndVal, sumVal, cnt;
};
node op(node x, node y) {
    return {
        x.sumIndVal + y.sumIndVal + x.cnt * y.sumVal,
        x.sumVal + y.sumVal,
        x.cnt + y.cnt
    };
}
```

Élément neutre

En plus de l'opération `node op(node x, node y)`, on va munir notre type d'un **élément neutre** e tel que

$$x \oplus e = e \oplus x = x \text{ pour tout } x$$

```
struct node { ... };
node op(node x, node y) { return {...}; }
node neutre() { return {...}; }
assert(op(x, neutre()) == x && op(neutre(), x) == x);
```

Élément neutre

En plus de l'opération `node op(node x, node y)`, on va munir notre type d'un **élément neutre** e tel que

$$x \oplus e = e \oplus x = x \text{ pour tout } x$$

- Minimum sur intervalle : $\min(x, e) = x$ pour tout x
- Somme sur intervalle
- Somme indice-valeur

Élément neutre

En plus de l'opération `node op(node x, node y)`, on va munir notre type d'un **élément neutre** e tel que

$$x \oplus e = e \oplus x = x \text{ pour tout } x$$

- Minimum sur intervalle : $e = +\infty$
- Somme sur intervalle
- Somme indice-valeur

Élément neutre

En plus de l'opération `node op(node x, node y)`, on va munir notre type d'un **élément neutre** e tel que

$$x \oplus e = e \oplus x = x \text{ pour tout } x$$

- Minimum sur intervalle : $e = +\infty$
- Somme sur intervalle : $(x + e) = x$ pour tout x
- Somme indice-valeur

Élément neutre

En plus de l'opération `node op(node x, node y)`, on va munir notre type d'un **élément neutre** e tel que

$$x \oplus e = e \oplus x = x \text{ pour tout } x$$

- Minimum sur intervalle : $e = +\infty$
- Somme sur intervalle : $e = 0$
- Somme indice-valeur

Élément neutre

En plus de l'opération `node op(node x, node y)`, on va munir notre type d'un **élément neutre** e tel que

$$x \oplus e = e \oplus x = x \text{ pour tout } x$$

- Minimum sur intervalle : $e = +\infty$
- Somme sur intervalle : $e = 0$
- Somme indice-valeur

```
struct node { int sumIndVal, sumVal, cnt; };
node op(node x, node y) {
    return {
        x.sumIndVal + y.sumIndVal + x.cnt * y.sumVal,
        x.sumVal + y.sumVal,
        x.cnt + y.cnt
    };
}
```

Élément neutre

En plus de l'opération `node op(node x, node y)`, on va munir notre type d'un **élément neutre** e tel que

$$x \oplus e = e \oplus x = x \text{ pour tout } x$$

- Minimum sur intervalle : $e = +\infty$
- Somme sur intervalle : $e = 0$
- Somme indice-valeur $e = \{0, 0, 0\}$

```
struct node { int sumIndVal, sumVal, cnt; };
node op(node x, node y) {
    return {
        x.sumIndVal + y.sumIndVal + x.cnt * y.sumVal,
        x.sumVal + y.sumVal,
        x.cnt + y.cnt
    };
}
node neutre() { return {0, 0, 0}; }
```

Exercice 1 : indice du minimum

Compléter le fichier `argmin.cpp` (op, `neutre` et boucle d'initialisation) pour trouver l'indice du minimum.

```
sh run.sh argmin.cpp
0 1 2 3 4 5 6 7 8 9
6 4 0 9 2 3 1 5 7 8
argmin 0 3 : valeur=0, indice=2
argmin 1 4 : valeur=0, indice=2
argmin 2 5 : valeur=0, indice=2
argmin 3 6 : valeur=2, indice=4
argmin 4 7 : valeur=1, indice=6
argmin 5 8 : valeur=1, indice=6
argmin 6 9 : valeur=1, indice=6
argmin 7 10 : valeur=5, indice=7
```

Exercice 2 : coder l'arbre binaire

Dans la suite, on notera l'opération multiplicativement xy au lieu de $x \oplus y$. Les feuilles contiennent les éléments, les noeuds internes i contiennent le produit des fils $2i$ et $2i + 1$.

<table>
<tr><td colspan="8">$1 = (a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8)$</td></tr>
<tr><td colspan="4">$2 = (a_1 a_2 a_3 a_4)$</td><td colspan="4">$3 = (a_5 a_6 a_7 a_8)$</td></tr>
<tr><td colspan="2">$4 = (a_1 a_2)$</td><td colspan="2">$5 = (a_3 a_4)$</td><td colspan="2">$6 = (a_5 a_6)$</td><td colspan="2">$7 = (a_7 a_8)$</td></tr>
<tr><td>(a_1)</td><td>(a_2)</td><td>(a_3)</td><td>(a_4)</td><td>(a_5)</td><td>(a_6)</td><td>(a_7)</td><td>(a_8)</td></tr>
</table>

$a_2 a_3 a_4 a_5 = (a_2)(a_3 a_4)(a_5)$.
Notez qu'on utilise l'associativité de l'opération : $(xy)z = x(yz)$.

- Rouge (disjoint) : retourne le neutre
- Vert (inclus) : retourne le contenu du noeud
- Bleu (à cheval) : appelle récursivement sur les fils et retourne le produit des résultats

Il n'y a que deux noeuds bleus sur chaque niveau de l'arbre donc $\mathcal{O}(\log n)$ appels récursifs.

Exercice 2 : coder l'arbre binaire

<table>
<tr><td colspan="8">$1 = a_2 a_3 a_4 a_5$</td></tr>
<tr><td colspan="4">$2 = a_2 a_3 a_4$</td><td colspan="4">$3 = a_5$</td></tr>
<tr><td colspan="2">$4 = a_2$</td><td colspan="2">$5 = a_3 a_4$</td><td colspan="2">$6 = a_5$</td><td colspan="2">$7 = e$</td></tr>
<tr><td>e</td><td>a_2</td><td></td><td></td><td>a_5</td><td>e</td><td></td><td></td></tr>
</table>

$a_2 a_3 a_4 a_5 = (a_2)(a_3 a_4)(a_5)$.
Notez qu'on utilise l'associativité de l'opération : $(xy)z = x(yz)$.

- Rouge (disjoint) : retourne le neutre
- Vert (inclus) : retourne le contenu du noeud
- Bleu (à cheval) : appelle récursivement sur les fils et retourne le produit des résultats

Il n'y a que deux noeuds bleus sur chaque niveau de l'arbre donc $\mathcal{O}(\log n)$ appels récursifs.

Exercice 2 : coder l'arbre binaire

<table>
<tr><td colspan="8">$1 = a_2 a_3 a_4 a_5$</td></tr>
<tr><td colspan="4">$2 = a_2 a_3 a_4$</td><td colspan="4">$3 = a_5$</td></tr>
<tr><td colspan="2">$4 = a_2$</td><td colspan="2">$5 = a_3 a_4$</td><td colspan="2">$6 = a_5$</td><td colspan="2">$7 = e$</td></tr>
<tr><td>e</td><td>a_2</td><td></td><td></td><td>a_5</td><td>e</td><td></td><td></td></tr>
</table>

- Rouge (disjoint) : retourne le neutre
- Vert (inclus) : retourne le contenu du noeud
- Bleu (à cheval) : appelle récursivement sur les fils et retourne le produit des résultats

Dans le fichier `segtree.cpp`, complétez les fonctions `affecter` et `_req`. Testez avec

```
sh run.sh indval.cpp
```

Modifications sur intervalle

Les types de modifications sont des familles $\mathcal{F}$ de fonctions `node f(node x)` closes par composition ($f, g \in \mathcal{F} \Rightarrow f \circ g \in \mathcal{F}$) et contenant l'identité ($\mathrm{id} \in \mathcal{F}$)

- Translation sur intervalle : $f_\Delta(x) = x + \Delta$
- Plafonnage sur intervalle : $f_p(x) = \min(x, p)$
- Set sur intervalle : $f_c(x) = c$

Modifications sur intervalle

Les types de modifications sont des familles $\mathcal{F}$ de fonctions `node f(node x)` closes par composition ($f, g \in \mathcal{F} \Rightarrow f \circ g \in \mathcal{F}$) et contenant l'identité ($\mathrm{id} \in \mathcal{F}$)

- Translation sur intervalle : $f_\Delta(x) = x + \Delta$: $f_\Delta \circ f_{\Delta'} = f_{\Delta+\Delta'}$ et $\mathrm{id} = f_0$
- Plafonnage sur intervalle : $f_p(x) = \min(x, p)$
- Set sur intervalle : $f_c(x) = c$

Modifications sur intervalle

Les types de modifications sont des familles $\mathcal{F}$ de fonctions `node f(node x)` closes par composition ($f, g \in \mathcal{F} \Rightarrow f \circ g \in \mathcal{F}$) et contenant l'identité ($\mathrm{id} \in \mathcal{F}$)

- Translation sur intervalle : $f_\Delta(x) = x + \Delta$: $f_\Delta \circ f_{\Delta'} = f_{\Delta+\Delta'}$ et $\mathrm{id} = f_0$
- Plafonnage sur intervalle : $f_p(x) = \min(x, p)$: $f_p \circ f_{p'} = f_{\min(p,p')}$ et $\mathrm{id} = f_{+\infty}$
- Set sur intervalle : $f_c(x) = c$

Modifications sur intervalle

Les types de modifications sont des familles $\mathcal{F}$ de fonctions `node f(node x)` closes par composition ($f, g \in \mathcal{F} \Rightarrow f \circ g \in \mathcal{F}$) et contenant l'identité ($\mathrm{id} \in \mathcal{F}$)

- Translation sur intervalle : $f_\Delta(x) = x + \Delta$: $f_\Delta \circ f_{\Delta'} = f_{\Delta+\Delta'}$ et $\mathrm{id} = f_0$
- Plafonnage sur intervalle : $f_p(x) = \min(x, p)$: $f_p \circ f_{p'} = f_{\min(p,p')}$ et $\mathrm{id} = f_{+\infty}$
- Set sur intervalle : $f_c(x) = c$: $f_c \circ f_{c'} = f_c$ et il faut ajouter l'identité $\mathrm{id}(x) = x$ manuellement

Toutes les fonctions $f \in \mathcal{F}$ doivent être compatibles avec le type `node` et l'opération associée :

$$f(x \oplus y) = f(x) \oplus f(y) \text{ pour tous } x, y$$

Exemple pour « max sur intervalle » et « translation sur intervalle » : $\max(x, y) + \Delta = \max(x + \Delta, y + \Delta)$

Moralement, on veut qu'on puisse connaître l'impact de la modification sur les éléments $a_l, a_{l+1}, \ldots, a_r$ directement à partir du noeud qui contient le produit $(a_l \oplus a_{l+1} \oplus \cdots \oplus a_r)$.

Ainsi, quand un noeud est inclus dans l'intervalle de la modification, on pourra y suspendre la modification.

Exercice 3 : somme et translation sur intervalle

Complétez `sumadd.cpp` pour résoudre somme et translation sur intervalle. Pourquoi stocker uniquement la somme ne suffit pas ?

```
struct node { int somme, longueur; }
// op, neutre
struct fun { int delta; };
// eval, composition, id
```

Testez avec

```
sh run.sh sumadd.cpp

add 10 to [2, 6)
.0 .1 .2 .3 .4 .5 .6 .7 .8 .9
.6 .4 10 19 12 13 .1 .5 .7 .8
somme 2 5 : somme=41, longueur=3
```

Propagation paresseuse

<table>
<tr><td colspan="8">$x_8 f(x_9) f(x_5) f(x_3), \mathrm{id}$</td></tr>
<tr><td colspan="4">$x_8 f(x_9) f(x_5), \mathrm{id}$</td><td colspan="4">$f(x_3), \mathrm{id}$</td></tr>
<tr><td colspan="2">$x_8 f(x_9), \mathrm{id}$</td><td colspan="2">$f(x_5), \mathrm{id}$</td><td colspan="2">x_6, f</td><td colspan="2">x_7, f</td></tr>
<tr><td>$\emptyset$</td><td>$f(x_9), \mathrm{id}$</td><td>x_{10}, f</td><td>x_{11}, f</td><td>$\emptyset$</td><td>$\emptyset$</td><td>$\emptyset$</td><td>$\emptyset$</td></tr>
</table>

Exercice 4 : propagation paresseuse

Compléter `push(i)` de sorte à ce que

x_i, f_i	
x_{2i}, f_{2i}	x_{2i+1}, f_{2i+1}

devienne

$f_i(x_i), \mathrm{id}$	
$x_{2i}, f_i \circ f_{2i}$	$x_{2i+1}, f_i \circ f_{2i+1}$

Reprenez la fonction `_req` de votre fichier `segtree.cpp` **en y ajoutant un push au début de la fonction**.

```
node _req(int iNoeud, int debNoeud, int finNoeud) {
    push(iNoeud);
    // copier-coller depuis segtree.cpp
}
```

Codez ensuite la fonction _app qui push au début puis

- Disjoint : ne rien faire
- Inclus : faire $f_i := f_{\mathrm{req}}$ puis push (pour que la valeur de x_i récupérée par le parent à cheval soit correcte)
- À cheval : appeler récursivement et recalculer $x_i = x_{2i} \cdot x_{2i+1}$

Testez en remplaçant `#include "lazy_bourrin.cpp"` par `#include "lazy.cpp"` dans `sumadd.cpp`