Algorithmique avancé

- ✔ L'arbre binaire indexé :l'arbre de Fenwick (Binary indexed tree)
- ✔ The segment tree
- ✔ Les Bitmasks

Par Mourad NOUAILI

enseignant d'informatique

Références :

- La chaîne youtube de Tushar Roy: https://www.youtube.com/channel/UCZLJf_R2sWyUtXSKiKlyvAw
- https://www.topcoder.com/community/data-science/data-science-tutorials/binary-indexed-trees/
- http://www.geeksforgeeks.org/segment-tree-set-1-sum-of-given-range/
- https://kartikkukreja.wordpress.com/2013/12/02/range-updates-with-bit-fenwick-tree/
- http://www.geeksforgeeks.org/binary-indexed-tree-range-update-range-queries/
- https://leetcode.com/articles/range-sum-query-mutable/
- http://bitdevu.blogspot.com/
- http://codeforces.com/blog/entry/18169
- http://graphics.stanford.edu/~seander/bithacks.html
- https://www.hackerrank.com
- https://www.codechef.com
- https://www.facebook.com/groups/1700330526904327/
- http://www.spoj.com/
- https://en.wikipedia.org/wiki/Inclusion%E2%80%93exclusion_principle
- https://leetcode.com/articles/recursive-approach-segment-trees-range-sum-queries-lazy-propagation/

Table des matières

L’arbre binaire indexé :l’arbre de Fenwick (Binary indexed tree (BIT))

Problème :

https://www.codechef.com/problems/MARBLEGF **(Il faut avoir un compte)**

Explication de l’exemple :

INPUT :
```
5 3
1000 1002 1003 1004 1005
S 0 2
G 0 3
S 0 2
```

A				
0	**1**	**2**	**3**	**4**
1000	1002	1003	1004	1005

- S 0 2 : Opération d’Addition de l’indice 0 à l’indice 2 : A[0] + A[1] + A[2] = 3005
- G 0 3 : Ajouter 3 à l’élément d’indice 0 tableau « A ».

- S 0 2 : Opération de sommation de l’indice 0 à l’indice 2 : A[0] + A[1] + A[2] = 3008

approche #1 : parcourir le tableau et cumuler la somme :

Pour chaque commande "S i j" (query) : parcourir le tableau de i à j est cumulé la somme. Dans le pire des cas la complexité temporelle est O(N).

Pour chaque commande "G i num" ou "T i num" (update), il suffit d’ajouter ou de soustraire "num" de "A[i]". La complexité temporelle est O(1)

Donc, la complexité temporelle de l’exécution des commandes est **O (Q * N)**

Code C++ : approche naïve : O(Q * N) :

Submission link : https://www.codechef.com/viewsolution/13063189
ideone : http://http://ideone.com/g3QXLc

```
/*
*Task: https://www.codechef.com/problems/MARBLEGF
*Lang: C++11
Time complexity: O(Q * N)
*/
#include <bits/stdc++.h>

using namespace std;

typedef long long ll;

vector<ll> A;

int N , Q;

//O(N)
ll sum (ll i , ll j) {
  ll s = 0;
  for (int k = i ; k <= j ; ++k) {
    s += A[k];
  }
  return s;
}

int main() {

  cin >> N >> Q ;

  A.resize(N);

  for (ll i = 0 ; i < N ; ++i) {
    ll ai;
    cin >> ai;
    A[i] = ai;
  }

  for (ll i = 0 ; i < Q ; ++i) {
    char qType;
    cin >> qType;
    switch (qType) {
      case 'G':{
      case 'T':
        ll i , m;
        cin >> i >> m;
        if (qType == 'T') m = -m ;
        A[i] += m;
        break;
      }
      case 'S': {
        ll i , j;
        cin >> i >> j;
        cout << sum(i,j) << '\n';
        break;
```

```
      }
    }
  }
  return 0;
}
```

La soumission de code sur le site donne un TLE:

approche #2 : utiliser un tableau commutatif des sommes

A				
0	**1**	**2**	**3**	**4**
1000	1002	1003	1004	1005

ACum					
0	**1**	**2**	**3**	**4**	**5**
0	1000	2002	3005	4009	5014

- The query command : "S i j" : il suffit de calculer : ACum[j+1] – Acum[i]
  S 0 2 = A[0] + A[1] + A[2] = ACum[3] – ACum[0] = 3008
  S 3 4 = A[3] + A[4] = ACum[5] – ACum[3] = 2009
  Cette commande se fait en O(1)
- The update commandes "G i num" ou "T i num" : il faut reconstruire tout le tableau cumulatif (O(N))
  G 0 3 ; A et ACum deviennent :

A				
0	**1**	**2**	**3**	**4**
1003	1002	1003	1004	1005

ACum					
0	**1**	**2**	**3**	**4**	**5**
0	1003	2005	3008	4012	5017

Code C++ : approche avec tableau cumutatif: O(Q * N) :

Submissin link: https://www.codechef.com/viewsolution/12454030
ideone : http://http://ideone.com/6FSlrx

```
//O(Q * N) : slow

#include <bits/stdc++.h>

using namespace std;

typedef long long ll;

int N , Q;

ll query (vector <ll> ACum, ll i,  ll j) {
  return  ACum[j+1] - ACum[i];
}

void update(vector <ll> & ACum, ll idx, ll val) {
  for (int i = idx+1 ; i <= N ; ++i)
    ACum[i] += val;
}

int main() {
  std::ios::sync_with_stdio(false);
  cin >> N >> Q ;

  vector<ll> ACum(N+1,0);

  for (ll i = 1 ; i <= N ; ++i) {
    ll ai;
    cin >> ai;
    ACum[i] = ACum[i-1] + ai;
  }
  for (ll i = 0 ; i < Q ; ++i) {
    char qType;
    cin >> qType;
    switch (qType) {
      case 'G':{
      case 'T':
        ll i , m;
        cin >> i >> m;
        if (qType == 'T') m = -m ;
        update(ACum,i,m);
        break;
      }
      case 'S': {
```

```
        ll i , j;
        cin >> i >> j;
        cout << query(ACum,i,j) << '\n';
        break;
      }
    }
  }
  return 0;
}
```

La soumission de code sur le site donne un TLE:

Approche #3 : Les arbres binaires indexés (les arbres de Fenwick) : Indexed binary tree (BIT) (Fenwick tree)

Une solution possible pour réduire le temps d'exécution est d'utiliser un arbre de binaire indexé appelé également : arbre de Fenwick.

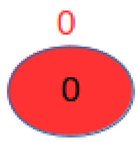
Construction de l'arbre Fenwick

Prenons un exemple :
Soit le tableau suivant sur lequel on veut appliquer les requêtes "S i j" et/ou "G i x" et/ou "T i x".

A																			
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**
4	4	7	4	12	-1	-9	3	1	3	0	5	3	-8	3	1	3	5	1	5

La taille de l'arbre = taille de A + 1
Algorithmiquement, l'arbre est représenté sous forme d'un tableau initialisé à 0 :

tree																				
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0



Les nœuds fils, d'un nœud i, sont calculés comme suit :
pour chaque nœud i :

- Le 1er fils est le nœud i + 1.
- Les autre fils : **(complément à 2 de(i+k) & (i+k)) + (i+k), i+1 < k ≤ N**

pour remplir l'arbre on applique la formule :

$$tree[i] = \sum_{i=i-2^r+1}^{i} A[i]$$

avec r est la position du premier bit "1" à partir de la droite. Et A est le tableau initial, tel que A[0] = 0

A																				
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
0	4	4	7	4	12	-1	-9	3	1	3	0	5	3	-8	3	1	3	5	1	5

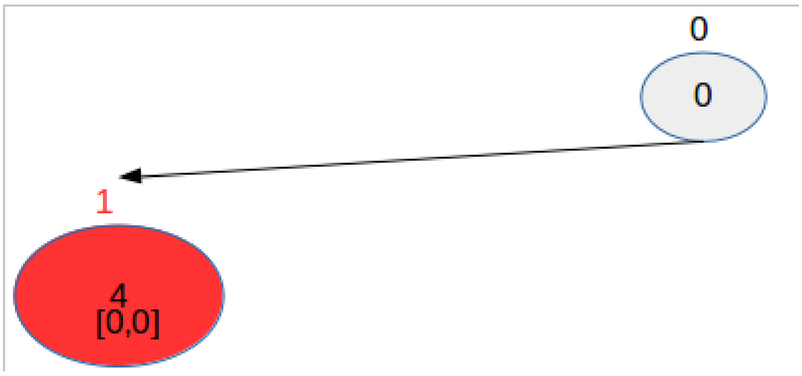
Les fils de "0" :

- Le premier fils 0 est 1.

1 = 01 => r = 0

$$tree[1]=\sum_{i=1-2^0+1}^{1} A[i]=\sum_{i=1}^{1} A[i]=A[1]=4$$

tree																				
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
0	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

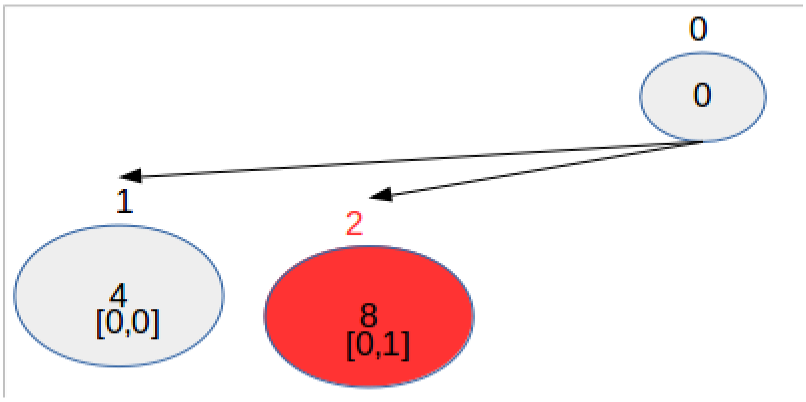


Les autres fils de 0 :

- complément à 2 de 1 = 1
- 1 & 1 = 1
- 1 + 1 = 10 = 2 en décimal

  r = 1

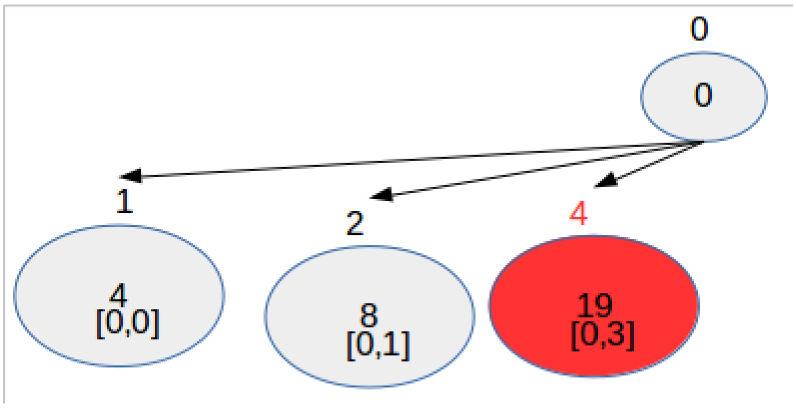
$$tree[2]=\sum_{i=2-2^1+1}^{2} A[i]=\sum_{i=1}^{2} A[i]=8$$



tree																				
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
0	4	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

- complément à 2 de 2 = 10 + 1 = 11
- 10 & 11 = 10
- 10 + 10 = 100 = 4

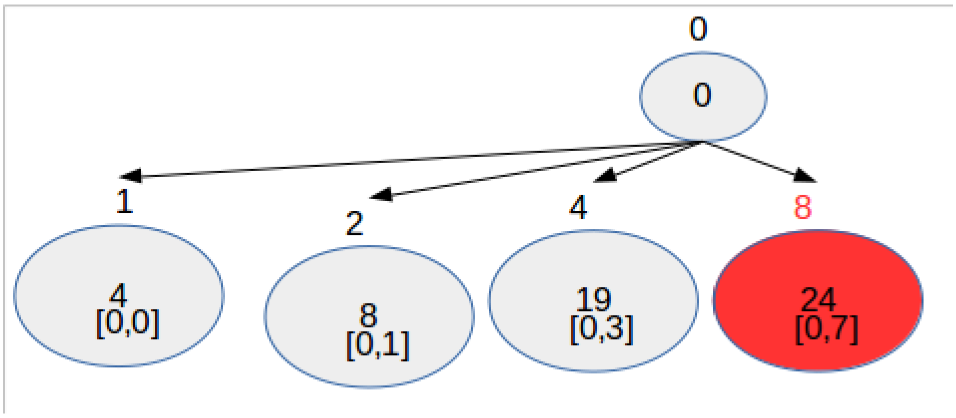
$$tree[4]=\sum_{i=4-2^2+1}^{4} A[i]=\sum_{i=1}^{4} A[i]=19$$



tree																				
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
0	4	8	0	19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

- complément à 2 de 4 = 011 + 1 = 100
- 100 & 100 = 100
- 100 + 100 = 1000 = 8

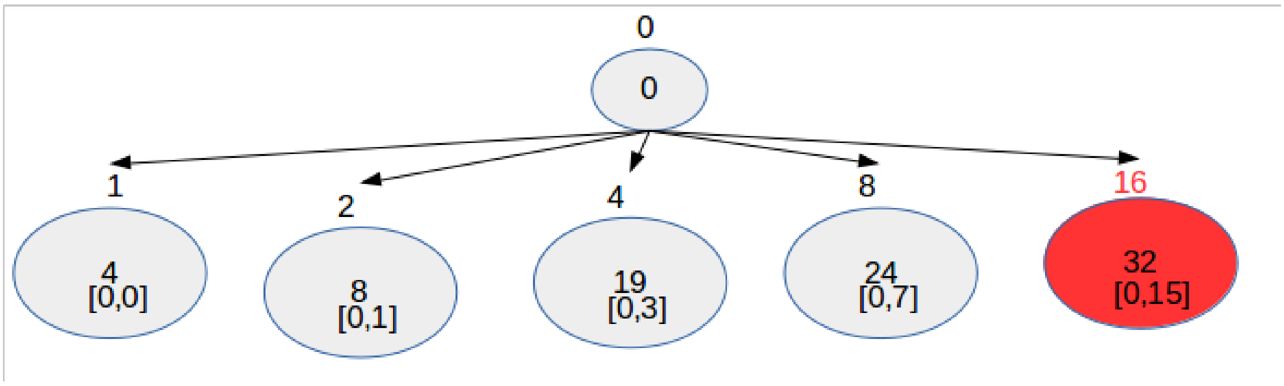
$$tree[8] = \sum_{i=8-2^4+1}^{8} A[i] = \sum_{i=1}^{8} A[i] = 24$$



tree																				
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
0	4	8	0	19	0	0	0	24	0	0	0	0	0	0	0	0	0	0	0	0

- complément à 2 de 8 = 0111 + 1 = 1000
- 1000 & 1000 = 1000
- 1000 + 1000 = 10000 = 16

$$tree[16]=\sum_{i=16-2^4+1}^{16} A[i]=\sum_{i=1}^{16} A[i]=32$$



tree																				
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
0	4	8	0	19	0	0	0	24	0	0	0	0	0	0	0	32	0	0	0	0

- complément à 2 de 16 = 01111 + 1 = 10000
- 10000 & 10000 = 10000
- 10000 + 10000 = 100000 = 32 > 20 (On arrête ici)

on passe maintenant aux fils de "1" :

le 1er fils de 1 est 2. (déjà pris)

les autres fils de 1 :

- complément à 2 de 2 = 10 + 1 = 11
- 10 & 11 = 10
- 10 + 10 = 100 = 4 (déjà pris)

- complément à 2 de 4 = 011 + 1 = 100
- 100 & 100 = 100
- 100 + 100 = 1000 = 8 (déjà pris)

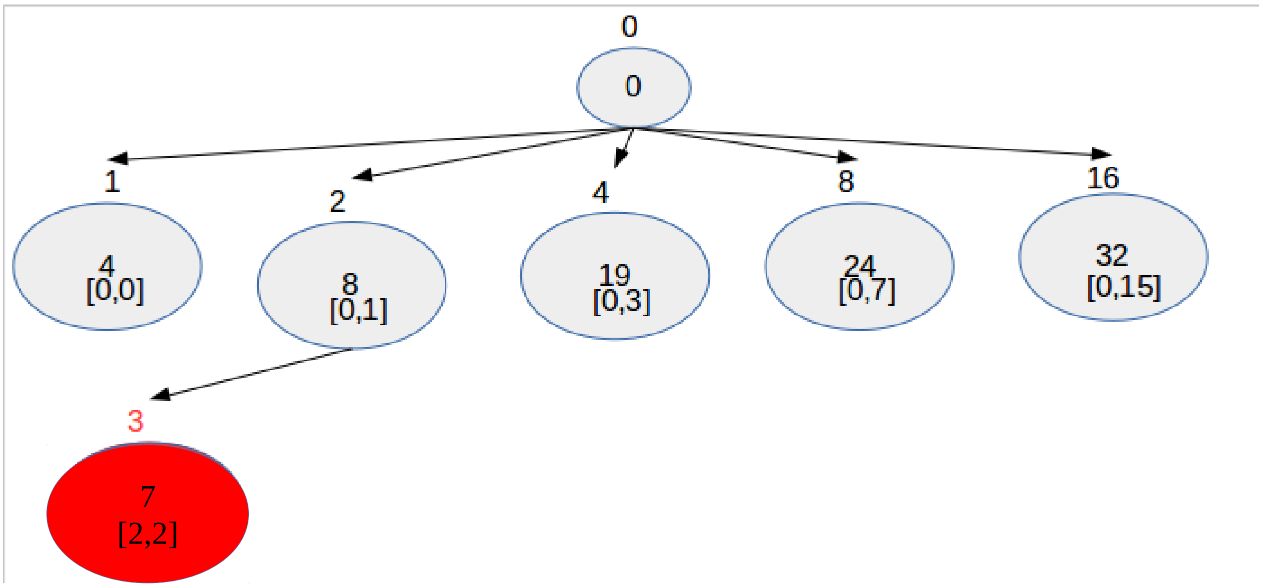
- complément à 2 de 8 = 0111 + 1 = 1000
- 1000 & 1000 = 1000
- 1000 + 1000 = 10000 = 16 (déjà pris)

- complément à 2 de 16 = 01111 + 1 = 10000
- 10000 & 10000 = 10000
- 10000 + 10000 = 100000 = 32 > 20 (On arrête ici)

Les fils de "2" :

le 1er fils de 2 est 3.

$$tree[3] = \sum_{i=3-2^0+1}^{3} A[i] = \sum_{i=3}^{3} A[i] = 7$$



tree																				
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
0	4	8	7	19	0	0	0	24	0	0	0	0	0	0	0	32	0	0	0	0

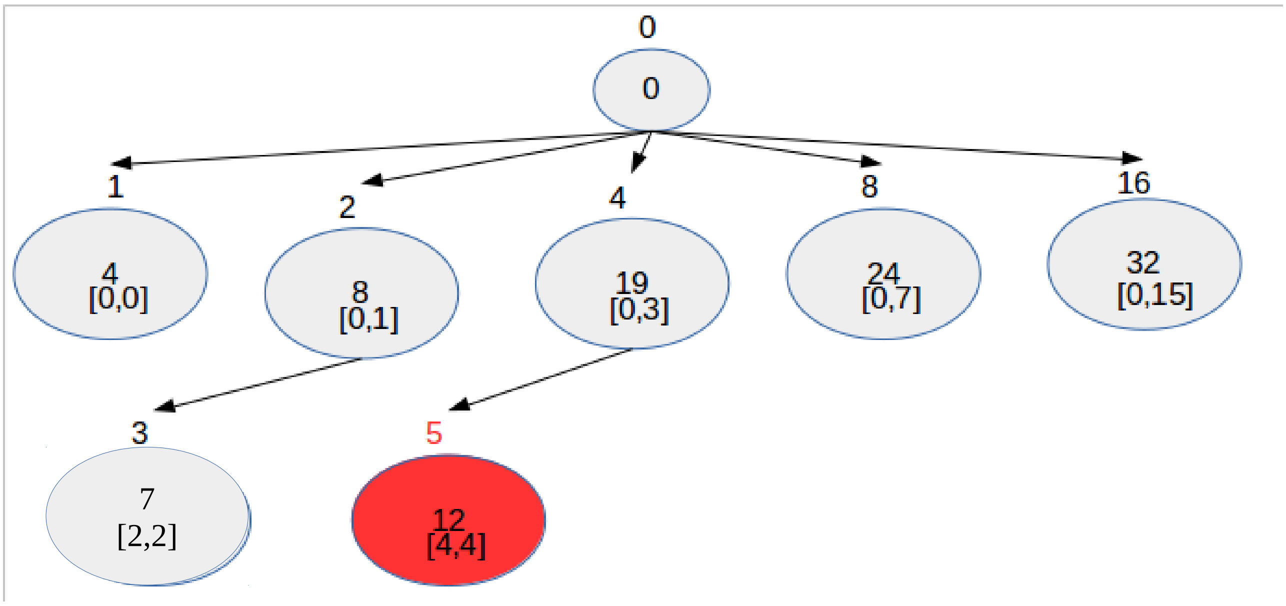
- complément à 2 de 3 = 00 + 1 = 01
- 11 & 01 = 01
- 11 + 01 = 100 = 4 (déjà pris)

- Si on continue avec le 4, on aura 8 , 16.

Les fils de "3"
le 1er fils de 3 est 4.
les autres sont 8 et 16.

Les fils de "4" :
le 1er fils de 4 est 5.

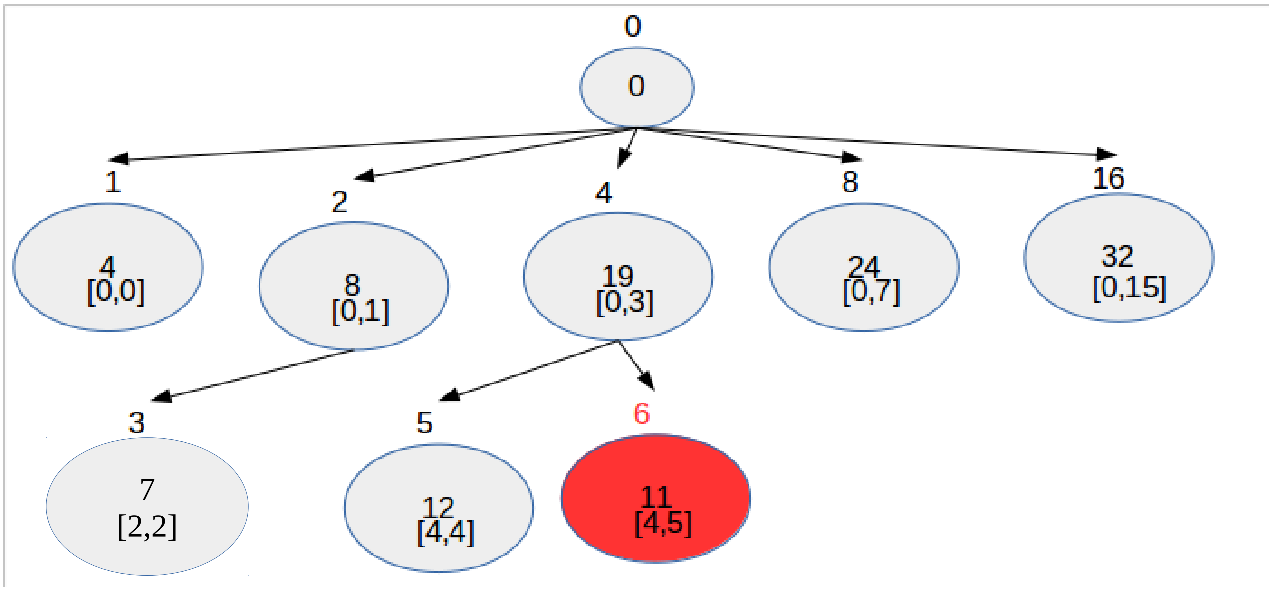
$$tree[5]=\sum_{i=5-2^0+1}^{5} A[i]=\sum_{i=5}^{5} A[i]=12$$



tree																				
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
0	4	8	7	19	12	0	0	24	0	0	0	0	0	0	0	32	0	0	0	0

- complément à 2 de 5 = 010 + 1 = 011
- 101 & 011 = 001
- 101 + 001 = 110 = 6

$$tree[6]=\sum_{i=6-2^1+1}^{6} A[i]=\sum_{i=5}^{6} A[i]=11$$



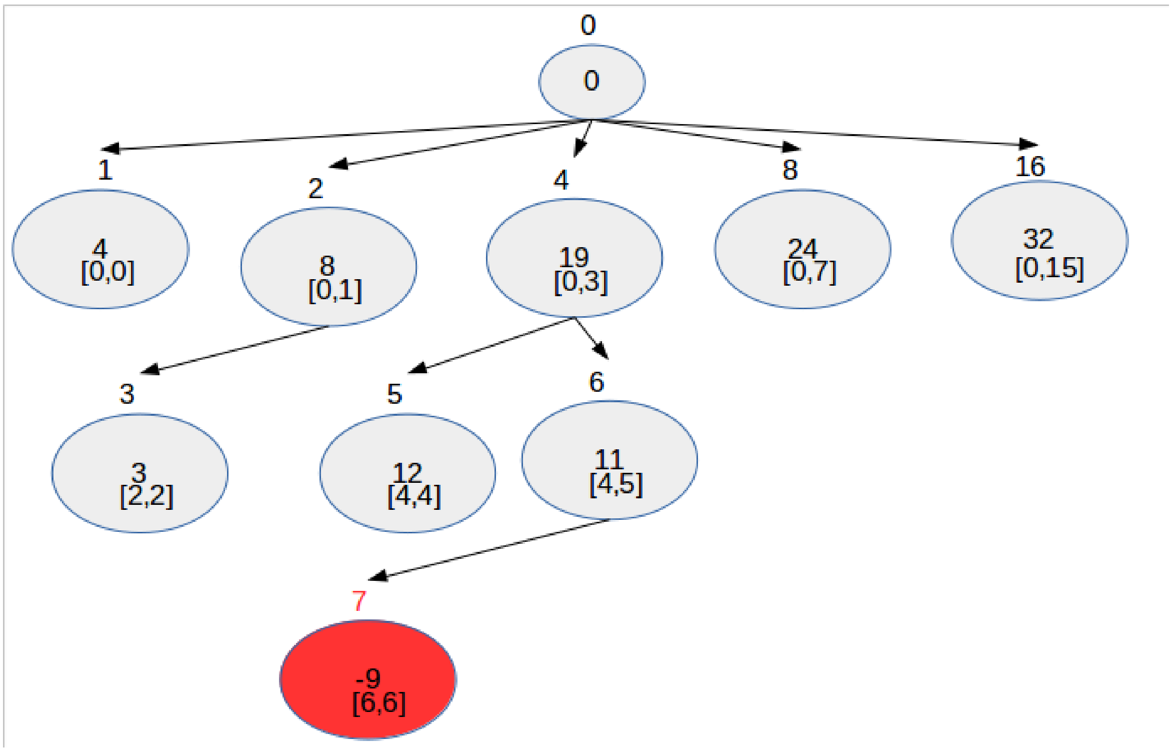
tree																				
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
0	4	8	7	19	12	11	0	24	0	0	0	0	0	0	0	32	0	0	0	0

- complément à 2 de 6 = 001 + 1 = 010
- 110 & 010 = 010
- 110 + 010 = 1000 = 8 (déjà pris)
- les autres fils sont pris.

Les fils de "5"
le 1er fils est 6
après nous avons le 8 et 16.

Les fils de "6" :
le 1er fils de 6 est 7.

$$tree[7]=\sum_{i=7-2^0+1}^{7} A[i]=\sum_{i=7}^{7} A[i]=-9$$



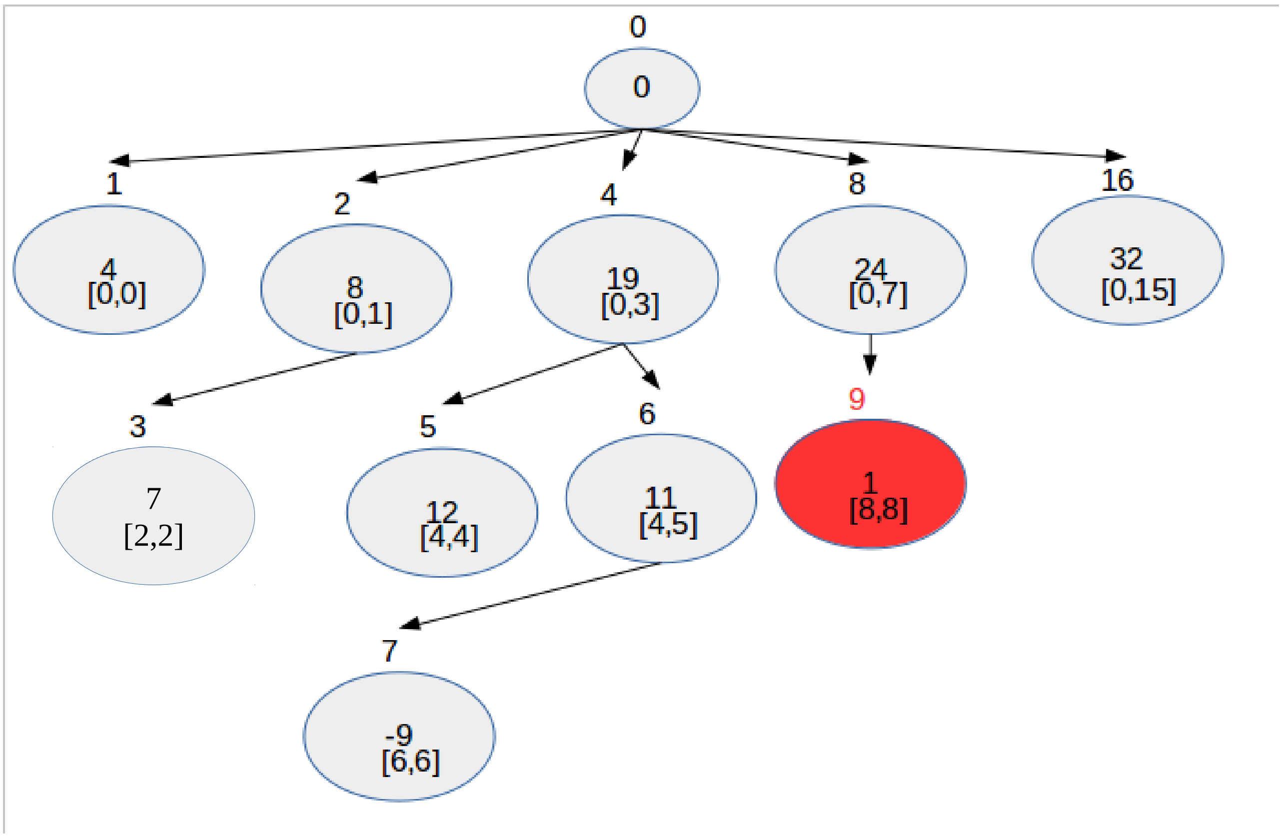
tree																				
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
0	4	8	7	19	12	11	-9	24	0	0	0	0	0	0	0	32	0	0	0	0

- complément à 2 de 7 = 000 + 1 = 001
- 111 & 001 = 001
- 111 + 001 = 1000 = 8 (déjà pris)
- les nœuds suivant sont pris aussi.

Les fils de "7" :
8 et 16

Les fils de "8" :
le 1[er] fils est 9

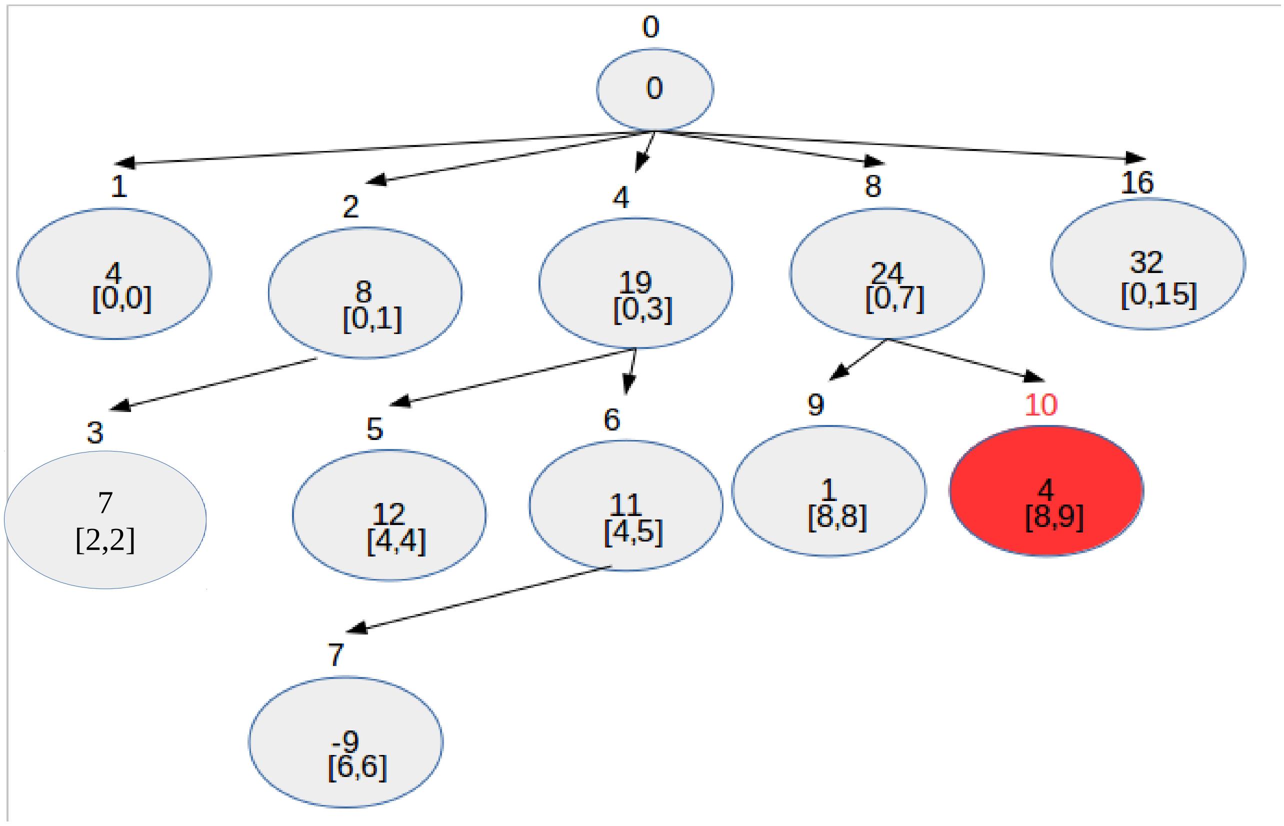
$$tree[9]=\sum_{i=9-2^0+1}^{9} A[i]=\sum_{i=9}^{9} A[i]=1$$



tree																				
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
0	4	8	7	19	12	11	-9	24	1	0	0	0	0	0	0	32	0	0	0	0

- complément à 2 de 9 = 0110 + 1 = 0111
- 1001 & 0111 = 0001
- 1001 + 0001 = 1010 = 10 en décimal

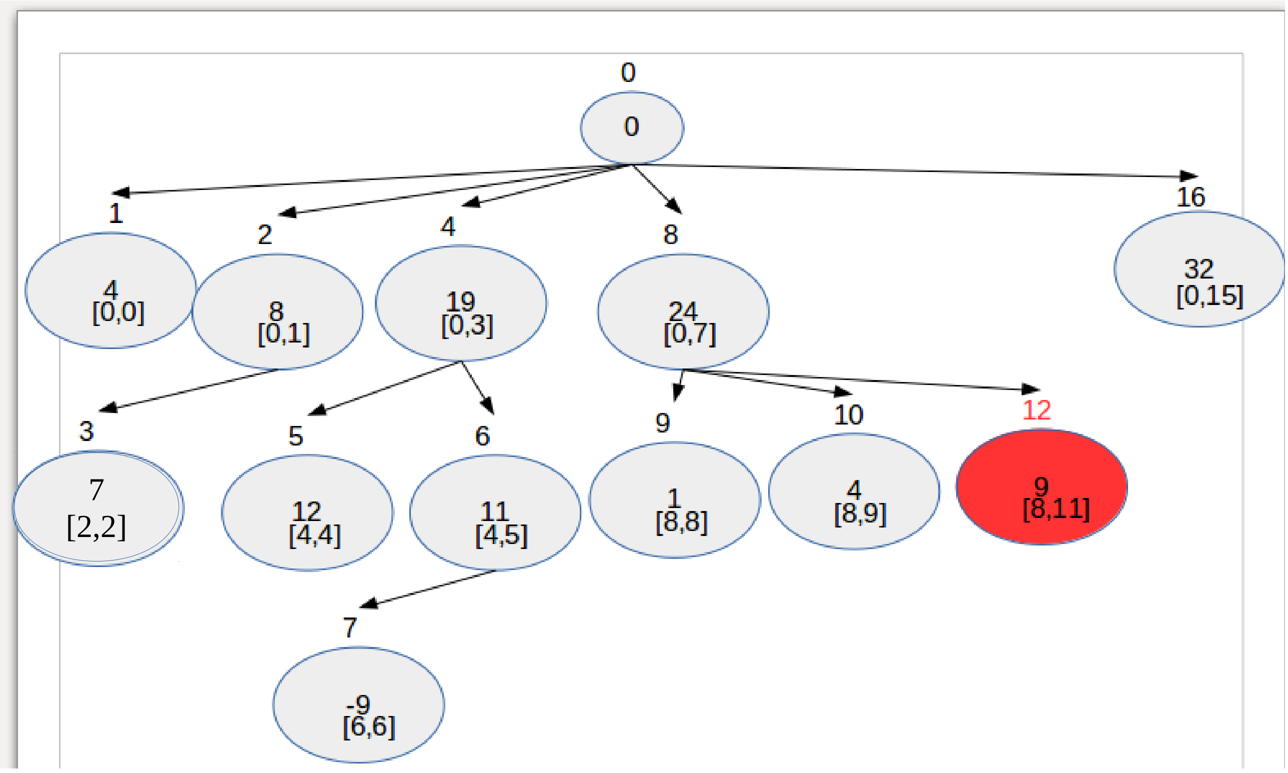
$$tree[10] = \sum_{i=10-2^1+1}^{10} A[i] = \sum_{i=9}^{10} A[i] = 4$$



tree																				
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
0	4	8	7	19	12	11	-9	24	1	4	0	0	0	0	0	32	0	0	0	0

- complément à 2 de 10 = 0101 + 1 = 0110
- 1010 & 0110 = 0010
- 1010 + 0010 = 1100 = 12 en décimal

$$tree[12] = \sum_{i=12-2^2+1}^{12} A[i] = \sum_{i=9}^{12} A[i] = 9$$



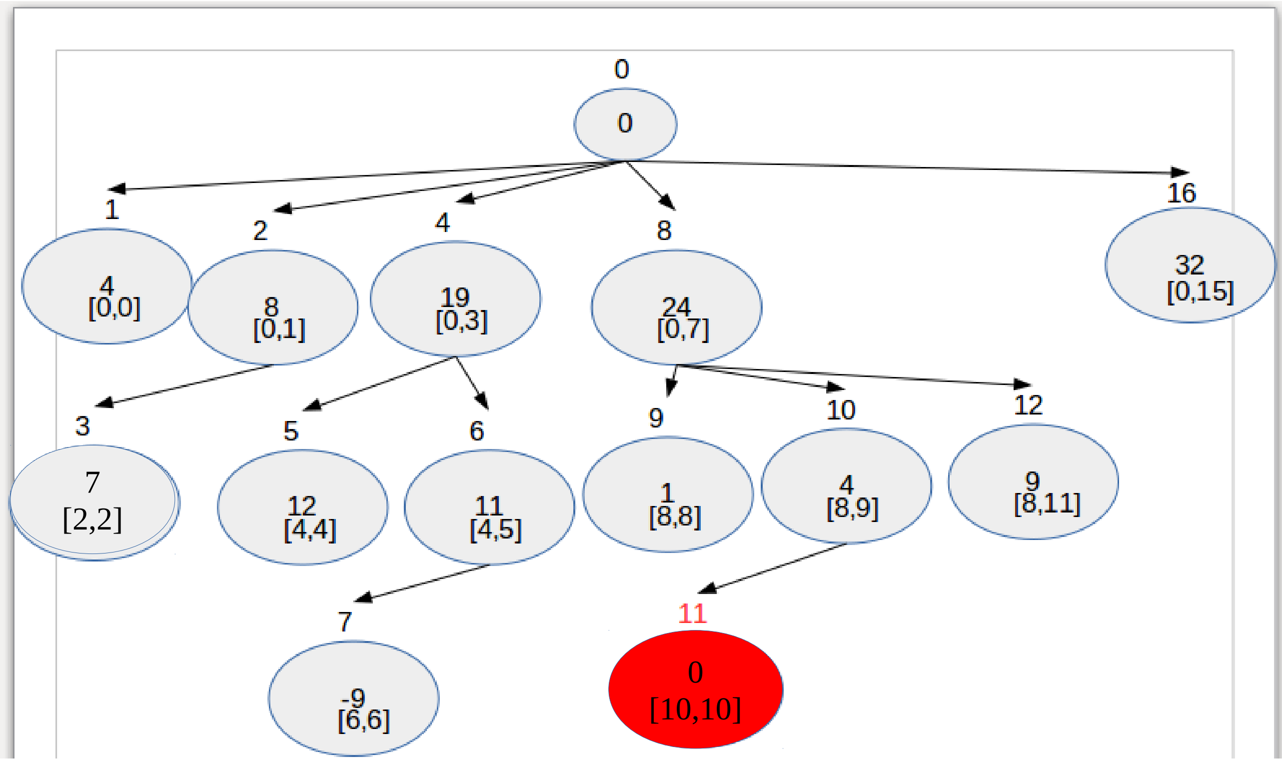
tree																				
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
0	4	8	7	19	12	11	-9	24	1	4	0	9	0	0	0	32	0	0	0	0

- complément à 2 de 12 = 0011 + 1 = 0100
- 1100 & 0100 = 0100
- 1100 + 0100 = 10000 = 16 en décimal (pris)

Les fils de "9" :
le 1^{er} fils est 10, puis 12 et 16. (déjà pris)

Les fils de "10" :
le 1^{er} fils est 11

$$tree[11] = \sum_{i=11-2^0+1}^{11} A[i] = \sum_{i=11}^{11} A[i] = 0$$



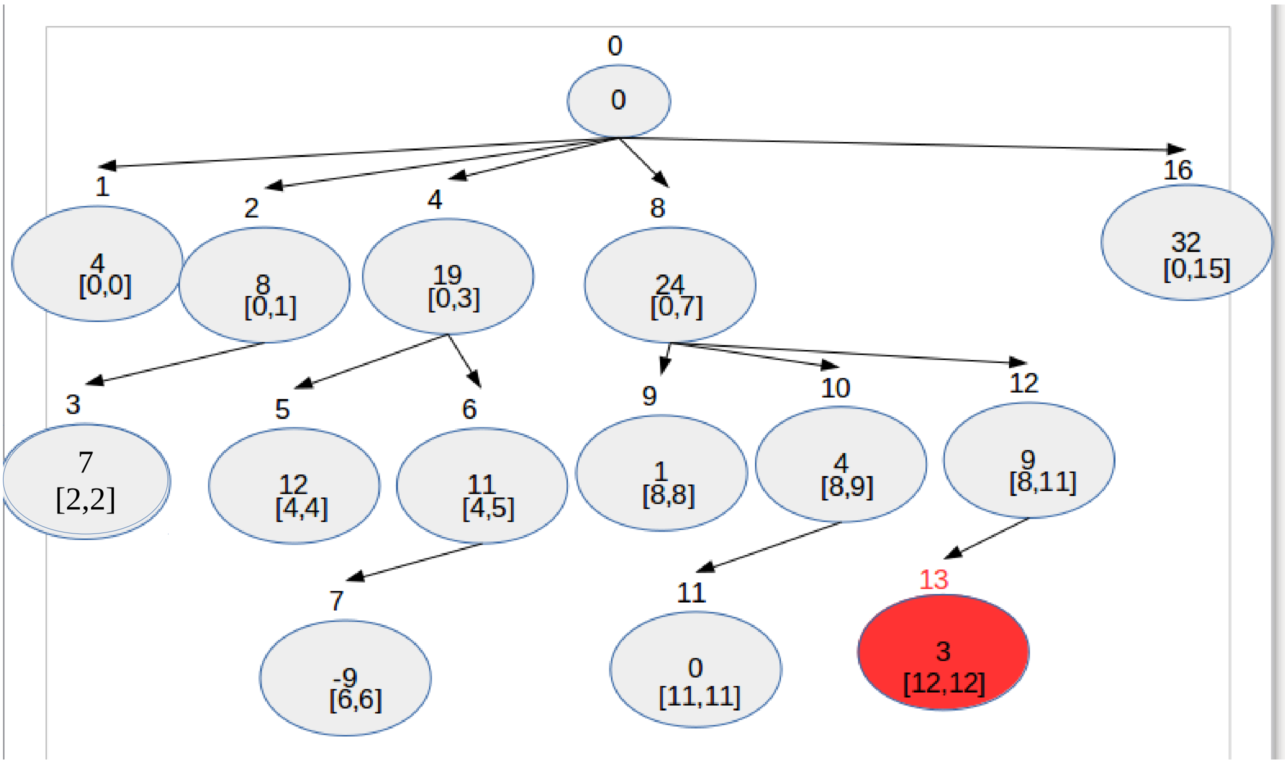
tree																				
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
0	4	8	7	19	12	11	-9	24	1	4	0	9	0	0	0	32	0	0	0	0

- complément à 2 de 11 = 0100 + 1 = 0101
- 1011 & 0101 = 0001
- 1011 + 0001 = 1100 = 12 (pris)

Les fils de "11" :
il sont tous pris

Les fils de "12" :
le 1er est 13

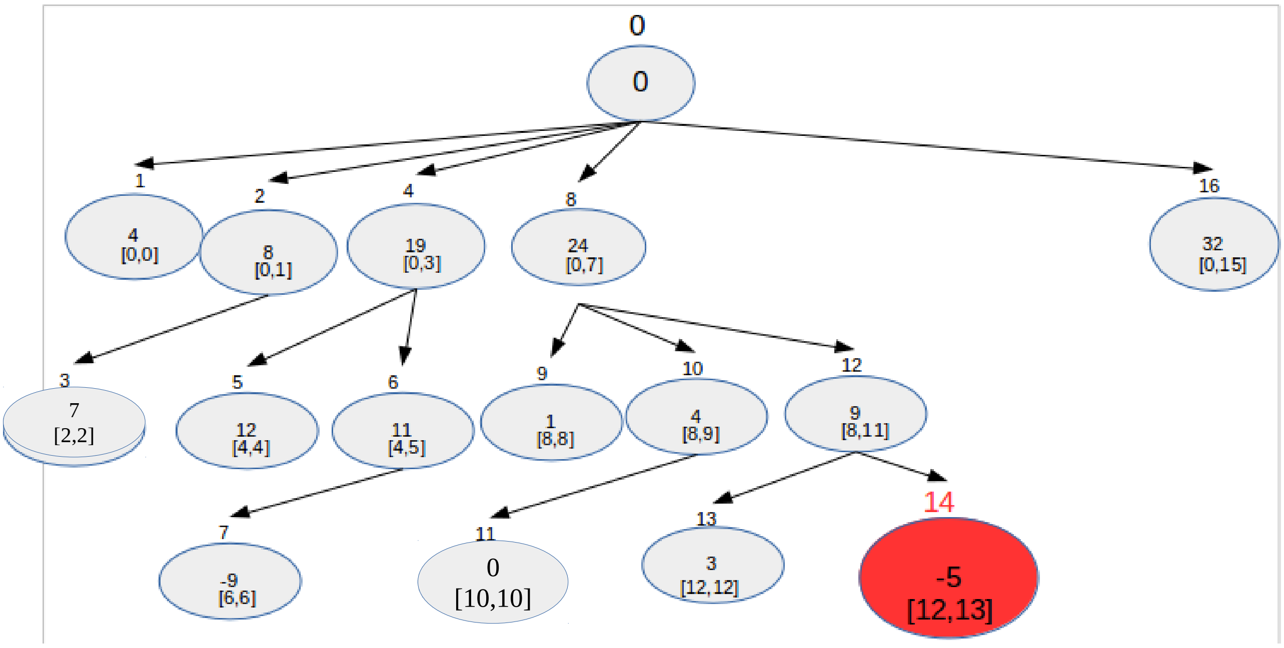
$$tree[13]=\sum_{i=13-2^0+1}^{13} A[i]=\sum_{i=13}^{13} A[i]=3$$



tree																				
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
0	4	8	7	19	12	11	-9	24	1	4	0	9	3	0	0	32	0	0	0	0

- complément à 2 de 13 = 0010 + 1 = 0011
- 1101 & 0011 = 0001
- 1101 + 0001 = 1110 = 14

$$tree[14]=\sum_{i=14-2^1+1}^{14} A[i]=\sum_{i=13}^{14} A[i]=-5$$



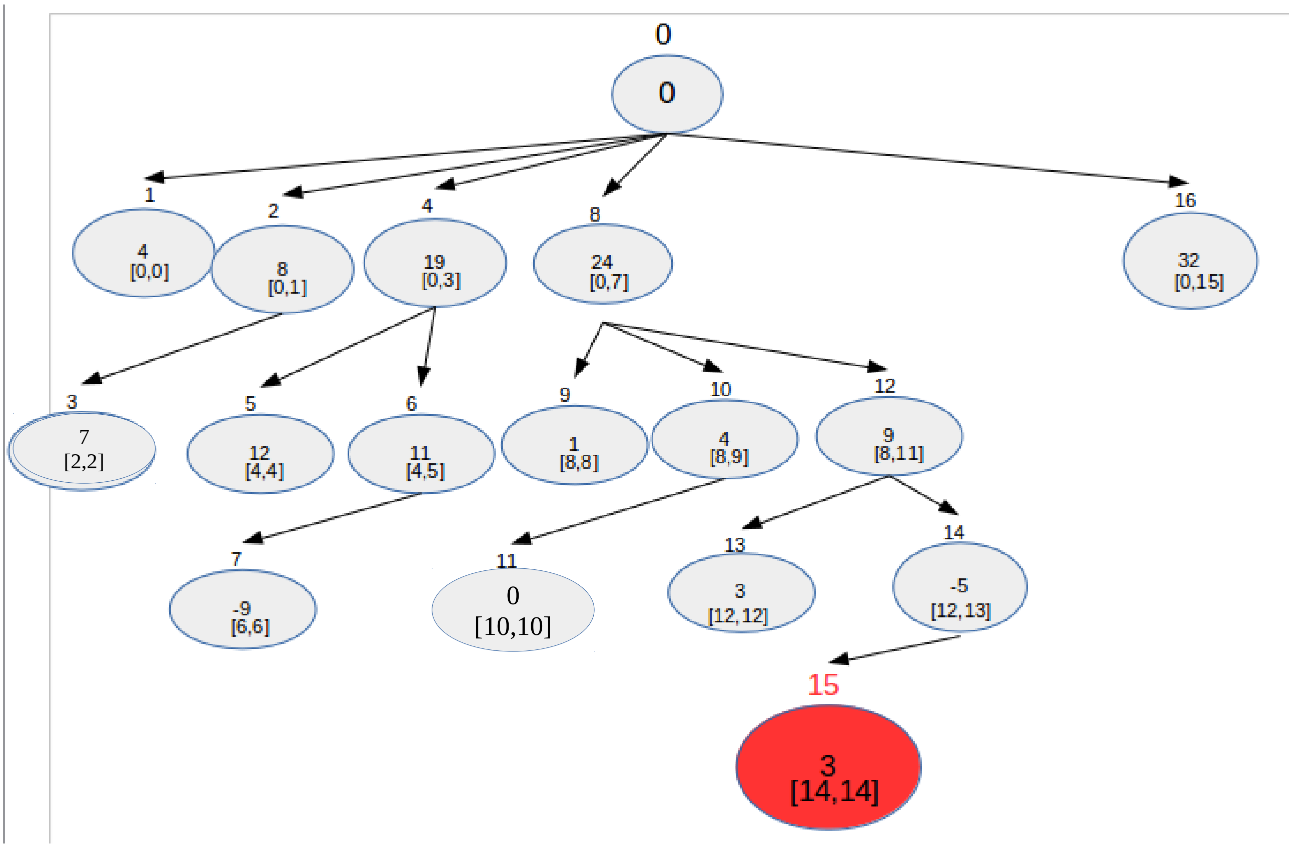
tree																				
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
0	4	8	7	19	12	11	-9	24	1	4	0	9	3	-5	0	32	0	0	0	0

- complément à 2 de 14 = 0001 + 1 = 0010
- 1110 & 0010 = 0010
- 1110 + 0010 = 10000 = 16 (pris)

Les fils de "13" :
14 et le suivants sont pris.

Les fils de "14" :
15 est le premier

$$tree[15]=\sum_{i=15-2^0+1}^{15} A[i]=\sum_{i=15}^{15} A[i]=3$$



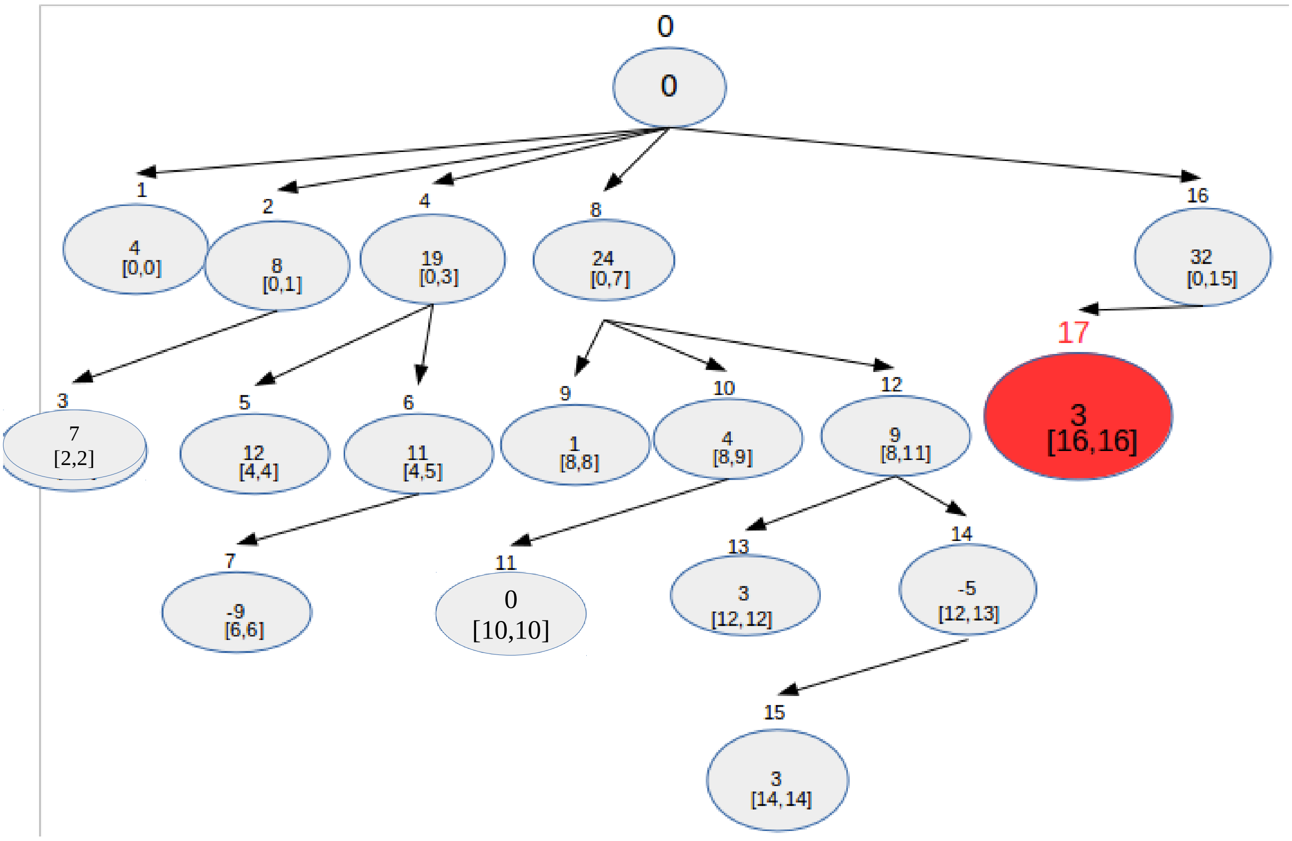
tree																				
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
0	4	8	7	19	12	11	-9	24	1	4	0	9	3	-5	3	32	0	0	0	0

- complément à 2 de 15 = 0000 + 1 = 0001
- 1111 & 0001 = 0001
- 1111 + 0001 = 10000 = 16 (pris)

Les fils de "15" :
le 1er est 16 (pris) puis 32 > 20

Les fils de "16" :
le 1er est 17

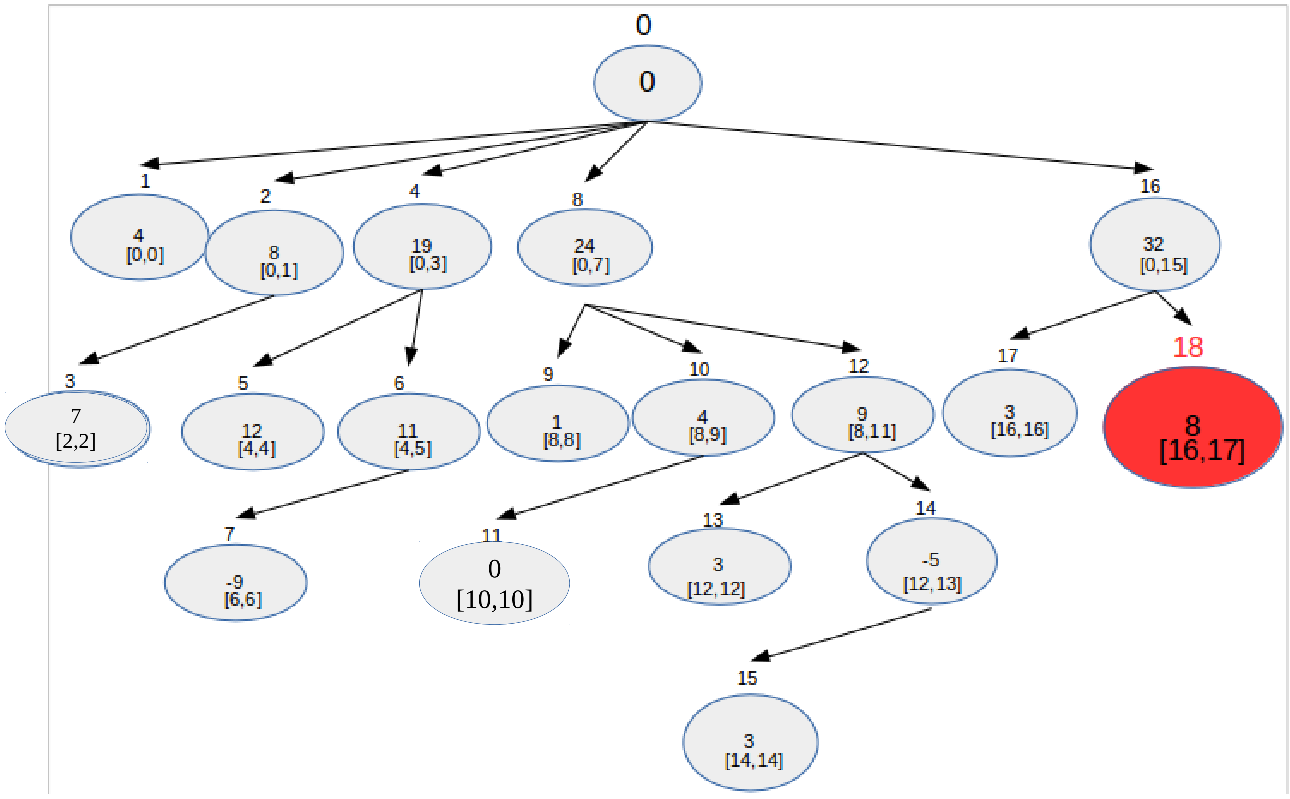
$$tree[17]=\sum_{i=17-2^0+1}^{17} A[i]=\sum_{i=17}^{17} A[i]=3$$



tree																				
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
0	4	8	7	19	12	11	-9	24	1	4	0	9	3	-5	3	32	3	0	0	0

- complément à 2 de 17 = 01110 + 1 = 01111
- 10001 & 01111 = 00001
- 10001 + 00001 = 10010 = 18

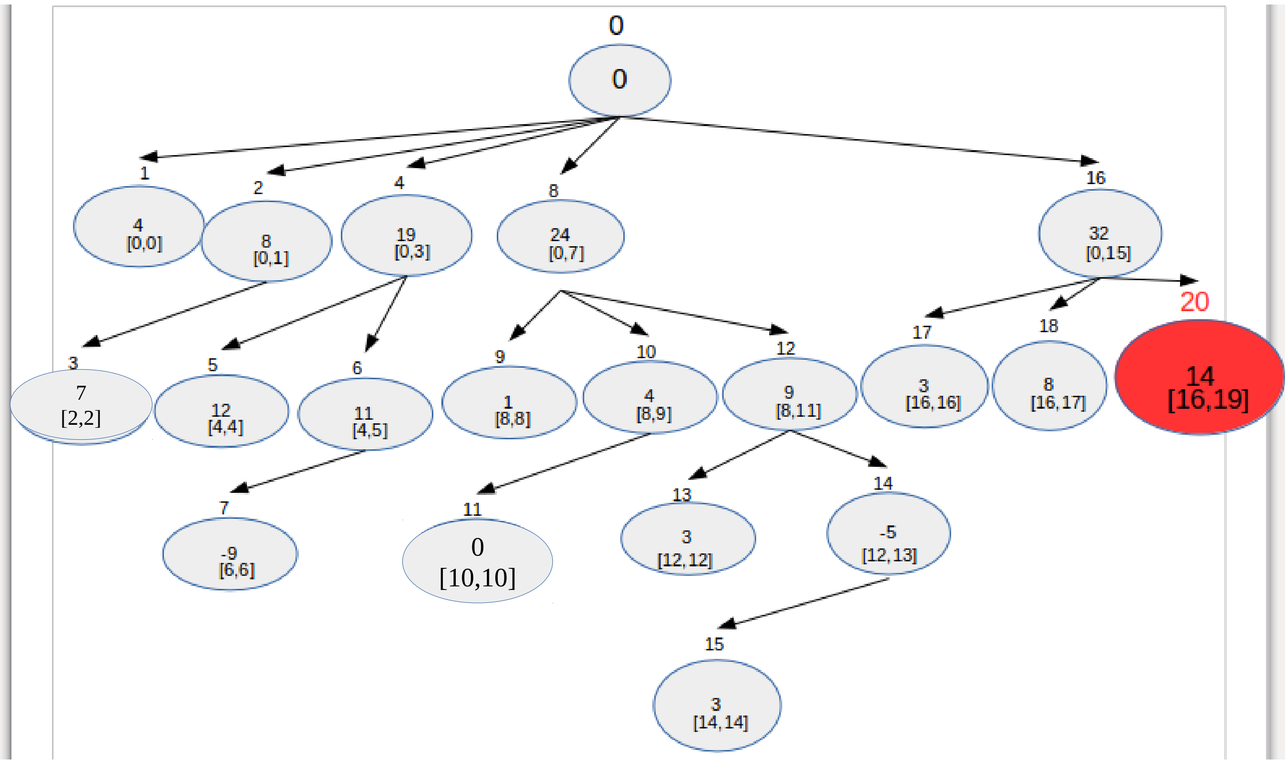
$$tree[18]=\sum_{i=18-2^1+1}^{18} A[i]=\sum_{i=17}^{18} A[i]=8$$



tree																				
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
0	4	8	7	19	12	11	-9	24	1	4	0	9	3	-5	3	32	3	8	0	0

- complément à 2 de 18 = 01101 + 1 = 01110
- 10010 & 01110 = 00010
- 10010 + 00010 = 10100 = 20

$$tree[20]=\sum_{i=20-2^2+1}^{20} A[i]=\sum_{i=17}^{20} A[i]=14$$

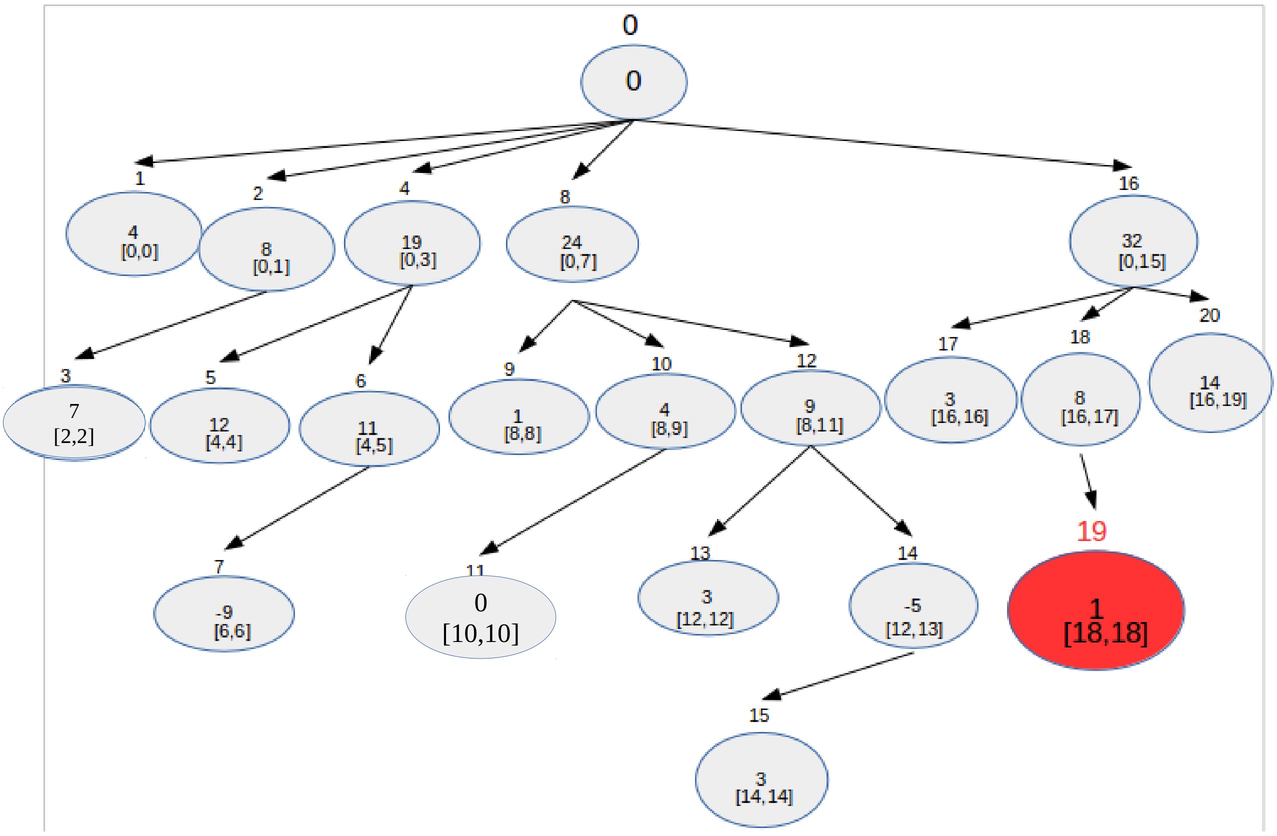


tree																				
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
0	4	8	7	19	12	11	-9	24	1	4	0	9	3	-5	3	32	3	8	0	14

Les fils de "17" :
18, 20 sont pris.

Les fils de "18" :
le 1er est 19

$$tree[19]=\sum_{i=19-2^0+1}^{19} A[i]=\sum_{i=19}^{19} A[i]=1$$



tree																				
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
0	4	8	7	19	12	11	-9	24	1	4	0	9	3	-5	3	32	3	8	1	14

- complément à 2 de 19 = 01100 + 1 = 01101
- 10011 & 01101 = 00001
- 10011 + 00001 = 10100 = 20 (pris)

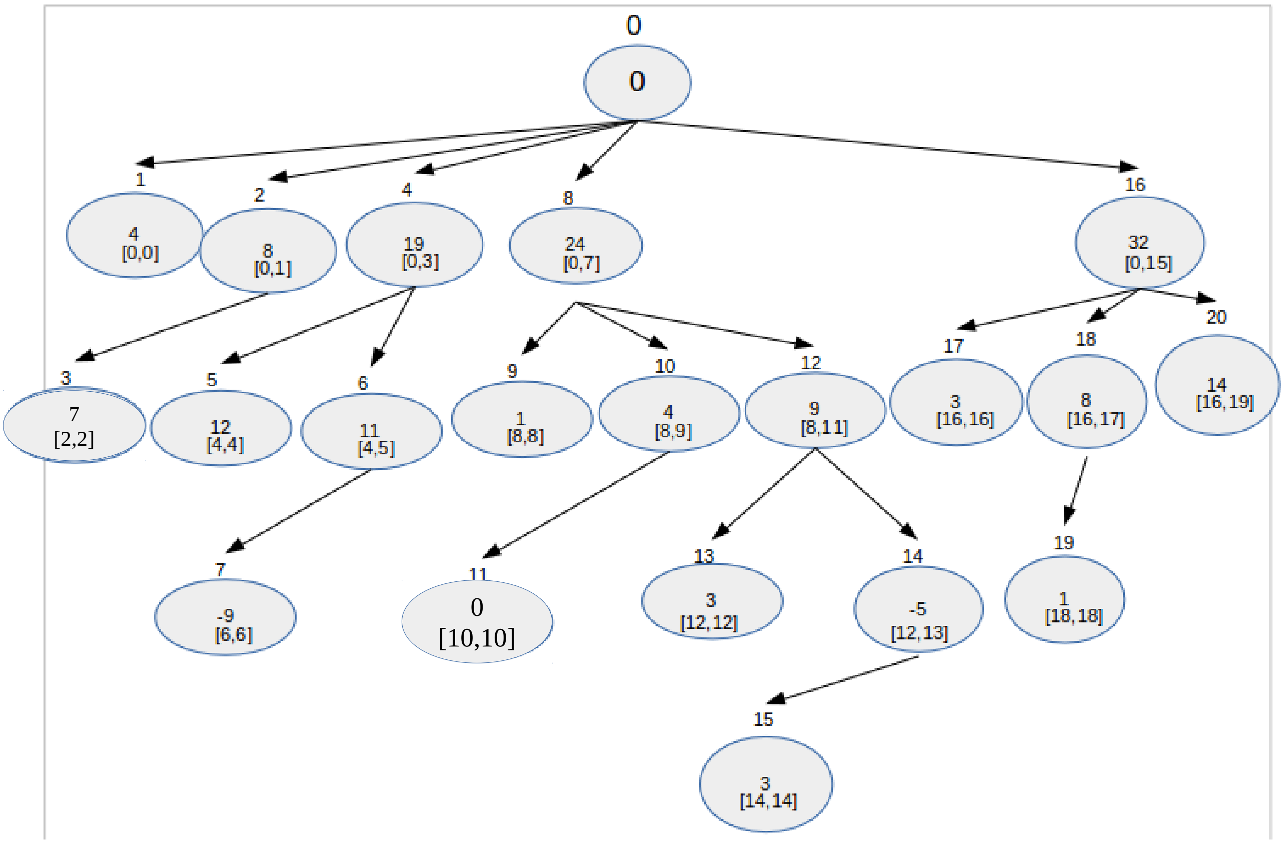
Les fils de "19" :
20 est pris

Les fils de "20" :
21 > 20

La Fenwick tree de tableau "A" :

A																			
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**
4	4	7	4	12	-1	-9	3	1	3	0	5	3	-8	3	1	3	5	1	5

est :

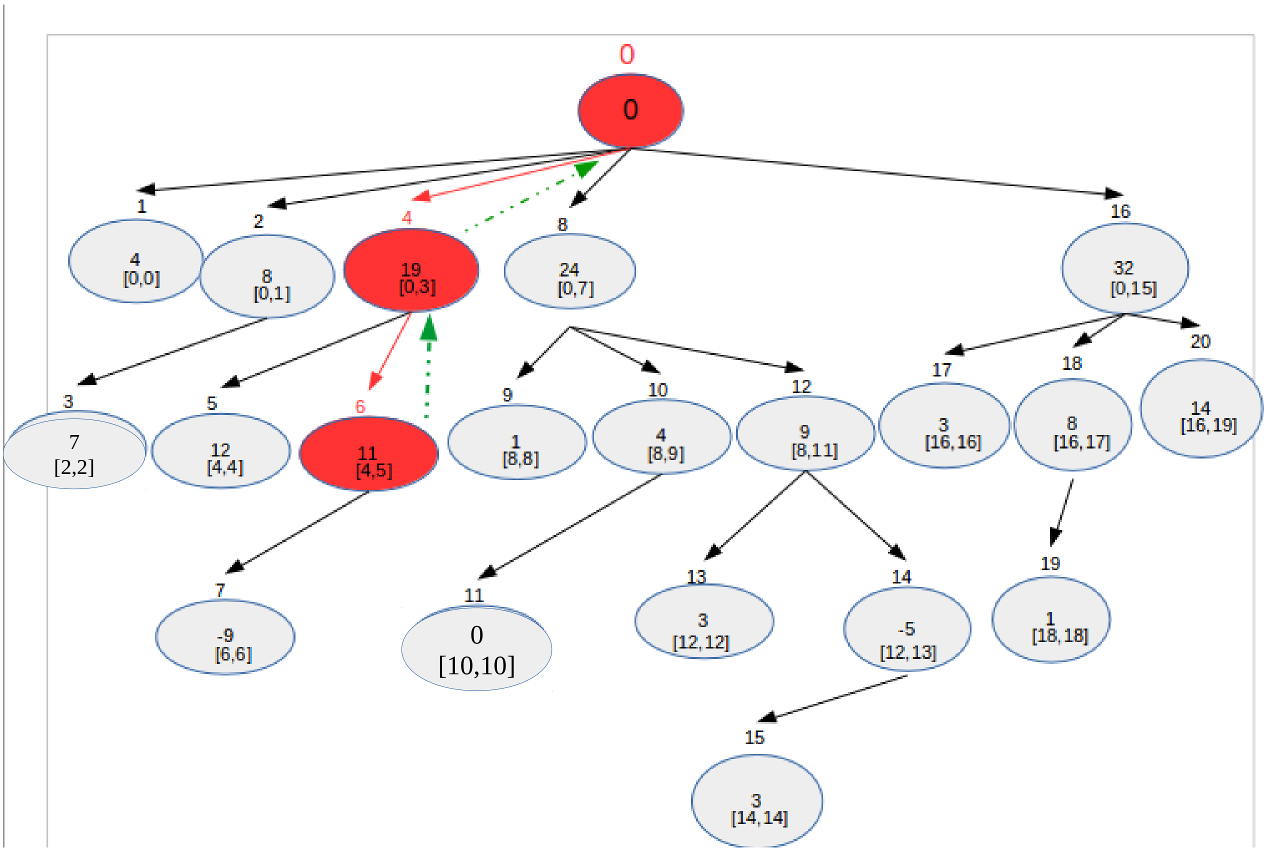


tree																				
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
0	4	8	7	19	12	11	-9	24	1	4	0	9	3	-5	3	32	3	8	1	14

Calculons la somme de 0 à 5

au lieu de faire la somme de A[0] à A[5], il suffit de à partir de l'arbre avec la manière suivante :

s	parent	Indice courant
0		
11	Complément à 2 de 6 = 1010 0110 & 1010 = 0010 0110 – 0010 = 0100 = 4	6
11+19	Complément à 2 de 4 = 1100 0100 & 1100 = 0100 0100 – 0100 = 0	4
30		0



On générale pour calculer la somme de [0,a], il suffit de commencer le calcul à partir du nœud "a" et en montant de parent parent jusqu'à la racine.
La complexité temporelle de ce calcul est O(log N)

la mise à jour de l'arbre

Modifions la valeur de A[4] par 20 (en ajoutant 8)

A																			
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**
4	4	7	4	20	-1	-9	3	1	3	0	5	3	-8	3	1	3	5	1	5

A																				
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
0	4	4	7	4	20	-1	-9	3	1	3	0	5	3	-8	3	1	3	5	1	5
Ancien tree																				
0	4	8	7	19	12	11	-9	24	1	4	0	9	3	-5	3	32	3	8	1	14
Nouveau tree																				
0	4	8	7	19	20	19	-9	32	1	4	0	9	3	-5	3	40	3	8	1	14

La mise à jour commence à partir du nœud "5", est se propage vers les nœuds suivants :

idx	tree[idx]	Nœuds suivant
5	12 + 8 = 20	Complément à 2 de 5 : 1011 0101 & 1011 = 0001 0101 + 0001 = 0110 = 6
6	11+8 = 19	Complément à 2 de 6 : 1010 0110 & 1010 = 0010 0110 + 0010 = 1000 = 8
8	24 + 8 = 32	Complément à 2 de 8 : 1000 1000 & 1000 = 1000 1000 + 1000 = 10000 = 16
16	32 + 8 = 40	Complément à 2 de 16 : 10000 10000 & 10000 = 100000 10000 + 10000 = 100000 = 32 > N = 20

Le fait de mettre à jour <u>une</u> case du tableau "A", se percute sur la Fenwick tree.
La complexité temporelle de l'update est O(log N).

Point update and range query (from 0 to…)

Jusqu’à maintenant nous avons vu, le calcule de la somme de 0 à "end" via la Fenwick tree : the range query. Et la mise à jour de à un point donnée : the point update.

Fonction C++ de point update :

```
void update(vector <int> & tree, int idx, int val) {
  while (idx <= N) {
    tree[idx] += val;
    idx += (idx & -idx);
  }
}
```

Fonction C++ de range query [0..idx]:

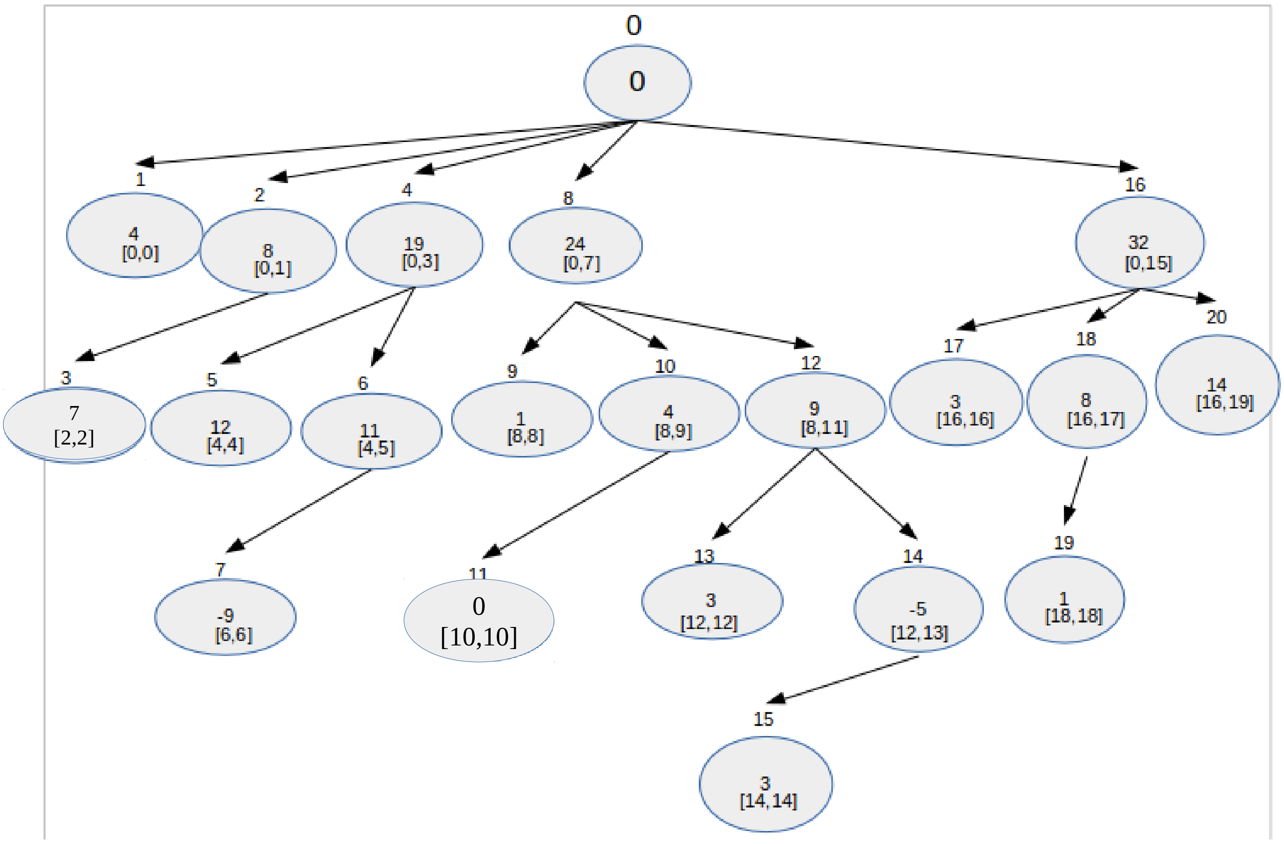
```
int query(vector <int> tree, int idx) {
  int s = 0;
  while (idx > 0) {
    s += tree[idx];
    idx -= (idx & -idx);
  }
  return s;
}
```

Point update and range query [a..b], 0 ≤ a < b < n

A																			
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**
4	4	7	4	12	-1	-9	3	1	3	0	5	3	-8	3	1	3	5	1	5

tree																				
0	4	8	7	19	12	11	-9	24	1	4	0	9	3	-5	3	32	3	8	1	14

Calculons la somme des éléments d'indices [9,15]
A[9] + A[10] + A[11] + A[12] + A[13] + A[14] + A[14] = 3 + 0 + 5 + 3 – 8 + 3 +1 = 7



A																			
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**
4	4	7	4	12	-1	-9	3	1	3	0	5	3	-8	3	1	3	5	1	5

Somme de [0,8]

Somme de [0,15]

Somme de [9,15] = somme [0,15] – somme [0,8]
Somme de [9,15] = query(16) – query(9)

en générale somme de [l,r] = query(r+1) – query(l)

Fonction C++ de point update :

```
void update(vector <int> & tree, int idx, int val) {
  while (idx <= N) {
    tree[idx] += val;
    idx += (idx & -idx);
  }
}
```

Fonction C++ de range query[0,b]:

```
int query(vector <int> tree, int idx) {
  int s = 0;
  while (idx > 0) {
    s += tree[idx];
    idx -= (idx & -idx);
  }
  return s;
}
```

Fonction C++ de range query[a,b]:

```
int rangeQuery(vector <int> tree, int l, int r) {
  return query(tree,r+1) - query(tree,l);
}
```

Point update and point query (A[i] = ?)

Fonction C++ de point query:

```
int readSingle(vector <int> tree, int idx){
  int v = tree[idx]; // sum will be decreased
  if (idx > 0){ // special case
      int z = idx - (idx & -idx); // make z first
      idx--; // idx is no important any more, so instead y, you can use idx
      while (idx != z){ // at some iteration idx (y) will become z
        v -= tree[idx];
// substruct tree frequency which is between y and "the same path"
      idx -= (idx & -idx);
    }
  }

  return v;
}
```

Code complet en C++ de l’exemple :

```
#include <bits/stdc++.h>

using namespace std;

int N;

void update(vector <int> & tree, int idx, int val) {
  while (idx <= N) {
    tree[idx] += val;
    idx += (idx & -idx);
  }
}

int query(vector <int> tree, int idx) {
  int s = 0;
  while (idx > 0) {
    s += tree[idx];
    idx -= (idx & -idx);
  }
  return s;
}

int rangeQuery(vector <int> tree, int l, int r) {
  return query(tree,r+1) - query(tree,l);
}

int readSingle(vector <int> tree, int idx){
  int v = tree[idx]; // sum will be decreased
  if (idx > 0){ // special case
      int z = idx - (idx & -idx); // make z first
      idx--; // idx is no important any more, so instead y, you can use idx
      while (idx != z){ // at some iteration idx (y) will become z
        v -= tree[idx];
// substruct tree frequency which is between y and "the same path"
      idx -= (idx & -idx);
    }
  }

  return v;
}

void display(vector <int> tree) {
  cout << "Fenwick tree:\n";
  for (int i = 0 ; i <= N ; ++i)
    cout << tree[i] << ' ';
  cout << '\n';
```

```
}


int main() {
  cout << "N = ?: ";
  cin >> N;

  vector <int> tree (N+1,0);

  cout << N << " entiers ?:\n";
  for (int i = 0 ; i < N ; ++i) {
    int val;
    cin >> val;
    update(tree,i+1,val);
  }

  cout << '\n';
  cout << "--------------------------------\n";
  cout <<"Point update & range query:\n";
  cout << "--------------------------------\n";

  cout << '\n';
  display(tree);
  cout << '\n';

  cout << "+Range query from [0..b]:\n";
  int pointQuery;
  cout << "--Somme de A[0] à A[?]: ";
  cin >> pointQuery;
  cout <<"--La somme de A[0] à A [" << pointQuery <<"] = " << query(tree,pointQuery+1) << '\n';

  cout << '\n';
  cout << "+Range query from [a..b]:\n";
  cout << "--Somme de A[?] à A[?]: ";
  int l,r;
  cin >> l >> r;
  cout <<"--La somme de A[" << l << "] à A [" << r <<"] = " << rangeQuery(tree,l,r) << '\n';

  int pointUpdate;
  cout << '\n';
  cout << "+Point update:\n";
  cout << "--indice du tableau A à mettre jour ?: ";
  cin >> pointUpdate;

  int val;
  cout << "--et par combien ?: ";
  cin >> val;
```

```
  update(tree,pointUpdate+1,val);
  cout << '\n';
  cout << "+Point query:\n";
  cout << "--A[" << pointUpdate <<"] = " << readSingle(tree,pointUpdate+1) << '\n';
  cout << '\n';
  display(tree);
  return 0;
}
```

```
N = ?: 20
20 entiers ?:
4 4 7 4 12 -1 -9 3 1 3 0 5 3 -8 3 1 3 5 1 5

-------------------------------
Point update & range query:
-------------------------------

Fenwick tree:
0 4 8 7 19 12 11 -9 24 1 4 0 9 3 -5 3 32 3 8 1 14

+Range query from [0..b]:
--Somme de A[0] à A[?]: 5
--La somme de A[0] à A [5] = 30

+Range query from [a..b]:
--Somme de A[?] à A[?]: 2 4
--La somme de A[2] à A [4] = 23

+Point update:
--indice du tableau A à mettre jour ?: 9
--et par combien ?: 5

+Point query:
--A[9] = 8

Fenwick tree:
0 4 8 7 19 12 11 -9 24 1 9 0 14 3 -5 3 37 3 8 1 14
```

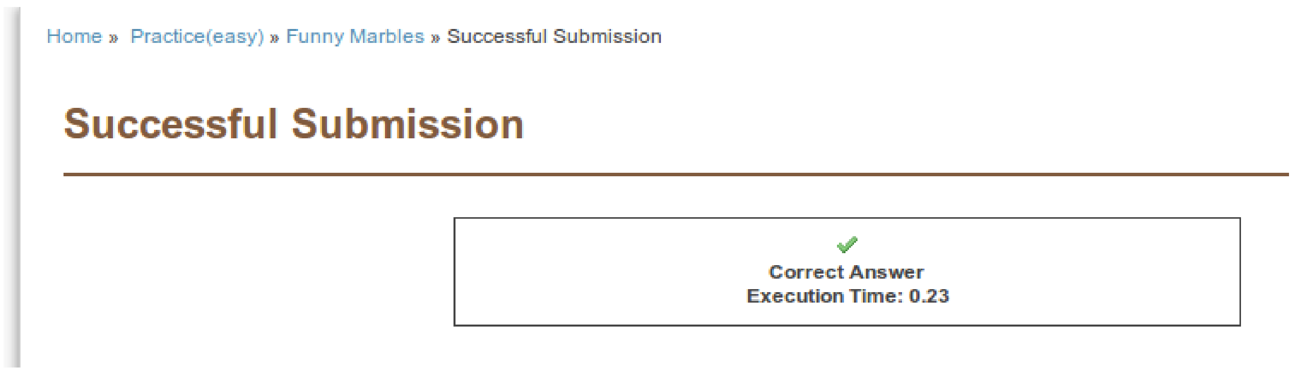
Le problème " https://www.codechef.com/problems/MARBLEGF" est de type point update, range query.

Solution du problème MARBLEGF de www.codechef.com

Lien de la soumission : https://www.codechef.com/viewsolution/12492408
ideone link : http://ideone.com/s93QO

```
/*
*Task: https://www.codechef.com/problems/MARBLEGF
*Lang: C++11
*Requirement knowledge: Fenwick tree (binary indexed tree)
Time complexity: O(Q * log N)
*/
#include <bits/stdc++.h>
using namespace std;

typedef long long ll;

ll N , Q;

//Fenwick code
//Point update
void update(vector <ll> & tree, ll idx, ll val) {
  while (idx <= N) {
      tree[idx] += val;
      idx += (idx & -idx);
    }
}

//Point query
ll query(vector <ll> & tree, ll idx){
    ll s = 0;
    while (idx > 0)  {
        s += tree[idx];
        idx -= (idx & -idx);
      }
    return s;
  }
//end Fenwick code.

int main() {
  std::ios::sync_with_stdio(false);
  cin >> N >> Q ;

  vector <ll> ft (N+1,0);

  for (ll i = 1 ; i <= N ; ++i) {
    ll ai;
```

```
    cin >> ai;
    update(ft,i,ai);
  }
  for (ll i = 0 ; i < Q ; ++i) {
    char qType;
    cin >> qType;
    switch (qType) {
      case 'G':{
      case 'T':
        ll i , m;
        cin >> i >> m;
        if (qType == 'T') m = -m ;
        update(ft,i+1,m);
        break;
      }

      case 'S': {
        ll i , j;
        cin >> i >> j;
        cout << query(ft,j+1) - query(ft,i)<< '\n';
        break;
      }
    }
  }
  return 0;
}
```



Homework :
http://www.spoj.com/problems/MSE06H/

[Solution : MSE06H-Editorial]()

Range update and point query (A[i] = ?)

Problème :
http://www.spoj.com/problems/UPDATEIT/

TLE solution
Submission : http://ideone.com/PCTSUQ

```
/*
*Task: http://www.spoj.com/problems/UPDATEIT/
    (Update the array !)
*Knowledge requirement: Fenwick tree (Binary indexed tree)
*Lang: C++
*Time complexity: O(t * log N  * (U * N  + q))
*SPOJ submission result: TLE
*/
#include <bits/stdc++.h>

using namespace std;

typedef long long li;

li N , U;


void update(vector <li> & tree, li idx, li val) {
  while (idx <= N) {
      tree[idx] += val;
      idx += (idx & -idx);
    }
}

li query(vector <li> & tree, li idx){
    li s = 0;
    while (idx > 0)  {
        s += tree[idx];
        idx -= (idx & -idx);
      }
    return s;
  }

li readSingle(vector <li> tree, li idx){
  li v = tree[idx];
  if (idx > 0) {
    li z = idx - (idx & -idx);
    idx--;
    while (idx != z){
      v -= tree[idx];
```

```
      idx -= (idx & -idx);
    }
  }
  return v;
}

void display(vector <li> tree) {
  cout << "Fenwink tree:\n";
  for (li i = 0 ; i <= N ; ++i)
    cout << tree[i] << ' ';
  cout << '\n';
}


int main() {
  std::ios::sync_with_stdio(false);
  li t;
  cin >> t;
  for (li ti = 0 ; ti < t ; ++ti) {
    cin >> N >> U;

    vector <li> tree (N+1,0);

    for (li ci = 0 ; ci < U ; ++ci ) {
      li l,r,val;
      cin >> l >> r >> val;
      for (li i = l+1 ; i <= r+1 ; ++i)
        update(tree,i,val);
    }

    li q;
    cin >> q;
    for (li qi = 0 ; qi < q ; ++qi) {
      li idx;
      cin >> idx;
      cout << readSingle(tree,idx+1) <<'\n';
    }
  }
  return 0;
}
```

ID		DATE	PROBLEM	RESULT	TIME	MEM	LANG
18570778		2017-01-14 01:43:32	Update the array !	time limit exceeded edit ideone it	-	3.6M	C++ 5

```
for (li i = l+1 ; i <= r+1 ; ++i)
    update(tree,i,val);
```

La complexité temporelle de l’update du tableau est O(N log N).
On peut la réduire en O (log N) en suivant la démarche suivante :

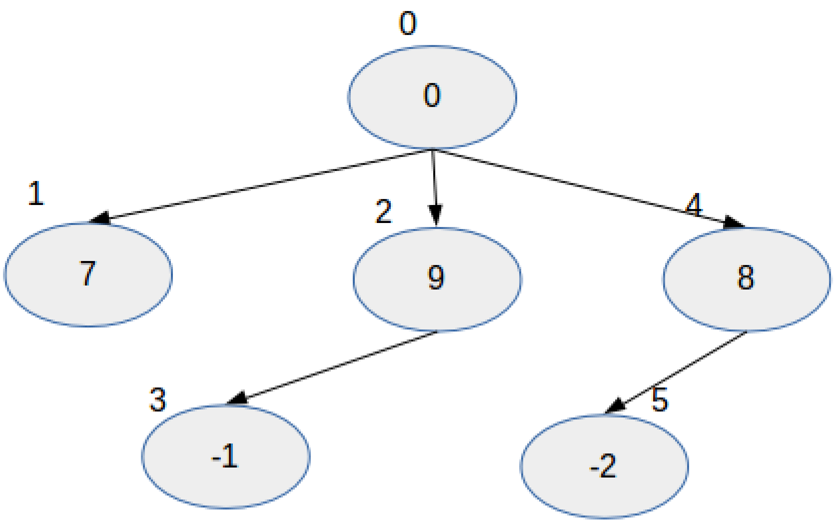
	A				
	0	**1**	**2**	**3**	**4**
0 1 7	*7*	*7*	0	0	0
2 4 6	7	7	*6*	*6*	*6*
1 3 2	7	*9*	*8*	*8*	6

l’astuce est que pour chaque update de type "l r v"n on ajoute "v" pour tous les i ≥ l et on réduit "-v" pour touts les i ≥ r+1, parce que :
update(tree,l,v) : touche tous les éléments d’indices [l,N], donc il faut supprimer -v aux éléments d’indices [r+1,N].

Illustration de l’exemple :

updates	**tree**					
	0	**1**	**2**	**3**	**4**	**5**
0 1 7	0	7	7	-7	0	
2 4 6	0			-1	6	
1 3 2	0		9		8	-2

tree					
0	**1**	**2**	**3**	**4**	**5**
0	7	9	-1	8	-2



Si on veut la
valeur de l’élément à l’indice "i", on peut utiliser la fonction suivante (pour la mise à jour):

Fonction C++ de point update :

```
void update(vector <int> & tree, int idx, int val) {
  while (idx <= N) {
    tree[idx] += val;
    idx += (idx & -idx);
  }
}
```

Fonction range update

```
void rangeUpdate(vector <int> & tree, int l, int r, int val) {
  update(tree,l, val);
  update(tree,r + 1, -val);
}
```

Puis, on applique la fonction "query" de [0,b] :

Fonction C++ de range query[0,b]:

```
int query(vector <int> tree, int idx) {
  int s = 0;
  while (idx > 0) {
    s += tree[idx];
    idx -= (idx & -idx);
  }
  return s;
}
```

accepted solution

Submission : http://ideone.com/KUUezH

```
/*
*Task: http://www.spoj.com/problems/UPDATEIT/
   (Update the array !)
*Knowledge requirement: Fenwick tree (Binary indexed tree)
*Lang: C++
*Time complexity: O(t * (U + q) * log N)
*SPOJ submission result: accpeted
*/
#include <bits/stdc++.h>

using namespace std;

typedef long long li;

li N , U;


void update(vector <li> & tree, li idx, li val) {
  while (idx <= N) {
      tree[idx] += val;
      idx += (idx & -idx);
    }
}

void rangeUpdate(vector <li> & tree, li l, li r, li val) {
  update(tree,l, val);
  update(tree,r + 1, -val);
}

li query(vector <li> & tree, li idx){
    li s = 0;
    while (idx > 0)  {
        s += tree[idx];
        idx -= (idx & -idx);
      }
    return s;
  }

void display(vector <li> tree) {
  cout << "Fenwink tree:\n";
  for (li i = 0 ; i <= N ; ++i)
    cout << tree[i] << ' ';
  cout << '\n';
  }
```

```
int main() {
  std::ios::sync_with_stdio(false);
  li t;
  cin >> t;
  for (li ti = 0 ; ti < t ; ++ti) {
    cin >> N >> U;

    vector <li> tree (N+1,0);

    for (li ci = 0 ; ci < U ; ++ci ) {
      li l,r,val;
      cin >> l >> r >> val;
      rangeUpdate(tree,l+1,r+1,val);
    }
    display(tree);
    li q;
    cin >> q;
    for (li qi = 0 ; qi < q ; ++qi) {
      li idx;
      cin >> idx;
      cout << query(tree,idx+1) <<'\n';
    }
  }
  return 0;
}
```

18570499		2017-01-13 23:52:10	Update the array !	**accepted** edit ideone it	0.24	3.5M	C++ 5

Range update and range query

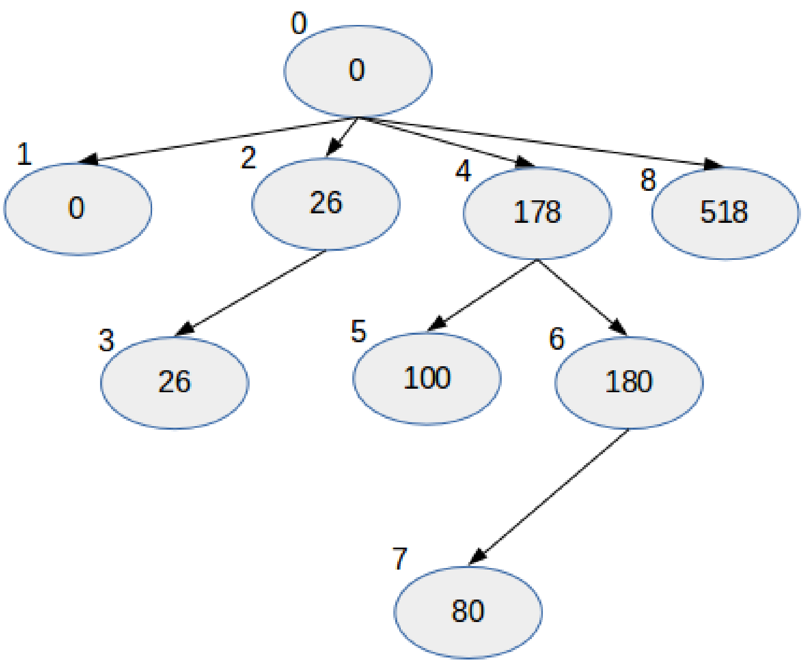
Problème :
http://www.spoj.com/problems/HORRIBLE/

```
Input:
1
8 6
0 2 4 26
0 4 8 80
0 4 5 20
1 8 8
0 5 7 14
1 4 8

Output:
80
508
```

Les opérations :
```
0 2 4 26
0 4 8 80
0 4 5 20
```

donnent :

tree								
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
0	0	26	26	178	100	180	80	518



Fenwick tree / segment tree / bitmasks

Pour faire la somme des éléments d’indices [l,r] :

```
li rangeQuery(vector <li> tree, li l, li r) {
  return query(tree,r) - query(tree,l-1);
}
```

TLE solution
Submission : http://ideone.com/I5sfoY

```
/*
*Task: http://www.spoj.com/problems/HORRIBLE/
   (HORRIBLE - Horrible Queries+)
*Knowledge requirement: Fenwick tree (Binary indexed tree)
*Lang: C++
*Time complexity: O(t * C * N * log N)
*SPOJ submission result: TLE
*/
#include <bits/stdc++.h>

using namespace std;

typedef long long li;

li N , C;


void update(vector <li> & tree, li idx, li val) {
  while (idx <= N) {
      tree[idx] += val;
      idx += (idx & -idx);
    }
}



li query(vector <li> & tree, li idx){
    li s = 0;
    while (idx > 0)  {
        s += tree[idx];
        idx -= (idx & -idx);
      }
    return s;
}

li rangeQuery(vector <li> tree, li l, li r) {
  return query(tree,r) - query(tree,l-1);
```

```
}

/*void display(vector <li> tree) {
  cout << "Fenwink tree:\n";
  for (int i = 0 ; i <= N ; ++i)
    cout << tree[i] << ' ';
  cout << '\n';
}*/

int main() {
  li t;
  cin >> t;
  for (li ti = 0 ; ti < t ; ++ti) {
    cin >> N >> C;

    vector <li> tree (N+1,0);

    for (li ci = 0 ; ci < C ; ++ci ) {
      int c;
      cin >> c;
      if (c == 0) {
        int p,q,v;
        cin >> p >> q >> v;
        for (li i = p ; i <= q; ++i)
          update(tree,i,v);
      }
      //display(tree);
      if (c == 1) {
        int p,q;
        cin >> p >> q;
        cout << rangeQuery(tree,p,q) << '\n';
      }
    }
  }
  return 0;
}
```

18585512		2017-01-16 16:57:18	Horrible Queries	time limit exceeded edit ideone it	-	4.2M	C++ 5

```
for (li i = p ; i <= q; ++i)
        update(tree,i,v);
```

L'update de l'arbre coûte N log N.

solution

On veut que l'update et le calcul de la somme soient fait en O(log N).

Le range sum est obtenu via les sommes cumulatifs (prefix sums).

Soit la situation suivante : on veut calculer la somme de 0 à p (sum[0,p]), avec $0 \le p < n$, après un range update entre l et r de v (add v to [l,r]).

Trois cas se présentent :

Cas #1 : $0 \le p < l$
sum[0,p] reste le même.

Cas #2 : $l \le p \le r$
sum[0,p] est incrémenté de v.p – v.(l-1)

Cas #3 : $r < p \le n$
sum[0,p] est incrémenté de v.r – v.(l-1)

donc, pour un indice "p" donné, on peut calculer la somme sum[1,p] en soustrayant une valeur "X" du p .A[p] :
Cas #1 : $0 \le p < l$
sum[0,p] => X = 0

=> sum[0,p] = p.A[p] - 0

Cas #2 : $l \le p \le r$
sum[0,p] est incrémenté de (v.p) – (v.(l-1)), X = v.(l-1)

sum[0,p] = p.A[p] – [v.(l-1)]

Cas #3 : $r < p \le n$
sum[0,p] est incrémenté de (v.r) – (v.(l-1)), X = -(v.r) + (v.(l-1))
=> sum[0,p] = p.A[p] - [-(v.r) + (v.(l-1))]

Pour trouver sum[0,p] :

- Maintenir un premier BIT (tree1) pour stocker les valeur du tableau (comme déjà vu dans range update/point query). query(tree1, p) donne la valeur de A[p]
- Maintenir un deuxième BIT (tree2), une fois query(tree2,p) invoqué donne la valeur de X selon les cas vu précédemment.

Pour ce-faire, on utilise deux arbres qui seront mis à jour comme suit :

Fonction C++ de point update :

```
void update(vector <int> & tree, int idx, int val) {
  while (idx <= N) {
    tree[idx] += val;
    idx += (idx & -idx);
  }
}
```

Fonction C++ de range update[l,r]:

```
void rangeUpdate(vector <li> & tree1,vector <li> & tree2, li l, li r, li val) {
  update(tree1,l, val);
  update(tree1,r + 1, -val);
  update(tree2,l,val*(l-1));
  update(tree2,r+1,-val*r);
}
```

Calcul de sum[l,r]

sum[l,r] = sum[0,r] – sum[0,l-1]
= [query(tree1,r) * r – query(tree2,r)] – [query(tree1,l-1) * (l-1) – query(tree2,(l-1)]

Fonctions C++ de rangeSum[l,r]:

```
int query(vector <int> & tree, int idx){
    int s = 0;
    while (idx > 0)  {
        s += tree[idx];
        idx -= (idx & -idx);
      }
    return s;
}

int sum(int idx, vector <int> & tree1, vector <int> & tree2) {
    return (query(tree1, idx) * idx) - query(tree2, idx);
}

int rangeSum(int l, int r, vector <int> & tree1, vector <int> &tree2){
    return sum(r, tree1, tree2) - sum(l-1, tree1, tree2);
}
```

Accpeted solution

Link of submission: http://www.spoj.com/submit/HORRIBLE/id=18598726
ideone: http://ideone.com/tYydCp

```
/*
*Task: http://www.spoj.com/problems/HORRIBLE/
    (HORRIBLE - Horrible Queries)
*Knowledge requirement: Fenwick tree (Binary indexed tree)
*Lang: C++
*Time complexity: O(t * C * log N)
*SPOJ submission result: accpeted
*/
#include <bits/stdc++.h>

using namespace std;

typedef long long li;

li N , C;


void update(vector <li> & tree, li idx, li val) {
  while (idx <= N) {
      tree[idx] += val;
      idx += (idx & -idx);
    }
}

void rangeUpdate(vector <li> & tree1,vector <li> & tree2, li l, li r, li val) {
  update(tree1,l, val);
  update(tree1,r + 1, -val);
  update(tree2,l,val*(l-1));
  update(tree2,r+1,-val*r);
}

li query(vector <li> & tree, li idx){
    li s = 0;
    while (idx > 0)  {
        s += tree[idx];
        idx -= (idx & -idx);
      }
    return s;
}

li sum(li idx, vector <li> & tree1, vector <li> & tree2) {
    return (query(tree1, idx) * idx) - query(tree2, idx);
}
```

```
li rangeSum(li l, li r, vector <li> & tree1, vector <li> &tree2){
    return sum(r, tree1, tree2) - sum(l-1, tree1, tree2);
}

  void display(vector <li> tree) {
    cout << "Fenwink tree:\n";
    for (int i = 0 ; i <= N ; ++i)
      cout << tree[i] << ' ';
    cout << '\n';
}

int main() {
  li t;
  cin >> t;
  for (li ti = 0 ; ti < t ; ++ti) {
    cin >> N >> C;

    vector <li> tree1 (N+1,0);
    vector <li> tree2 (N+1,0);

    for (li ci = 0 ; ci < C ; ++ci ) {
      int c;
      cin >> c;
      if (c == 0) {
        int p,q,v;
        cin >> p >> q >> v;
        rangeUpdate(tree1,tree2,p,q,v);
      }
      if (c == 1) {
        int p,q;
        cin >> p >> q;
        cout << rangeSum(p,q,tree1,tree2) << '\n';
      }
    }
  }
  return 0;
}
```

ID		DATE	PROBLEM	RESULT	TIME	MEM	LANG
18598726		2017-01-18 18:45:00	Horrible Queries	accepted edit ideone it	0.40	4.2M	C++ 5

The segment tree

Une autre structure de donnée qu’on peut utiliser est le "segment tree".

Ce type d’arbre permet de résoudre rapidement les requêtes de plage (range queries), tel que :

- Chercher le minimum dans une certaine plage d’éléments. (min range query)
- Chercher le maximum dans une certaine plage d’éléments. (max range query)
- Retourne la somme d’une certaine plage d’entiers. (sum range query)

Revenons au problème : MARBLEGF

https://www.codechef.com/problems/MARBLEGF

input				
0	1	2	3	4
1000	1002	1003	1004	1005

Construction du segment tree

On continue à diviser le tableau par 2, jusqu’à on obtient des tableaux d’1 seul élément:

0	1	2
1000	1002	1003

3	4
1004	1005

0	1
1000	1002

2
1003

3
1004

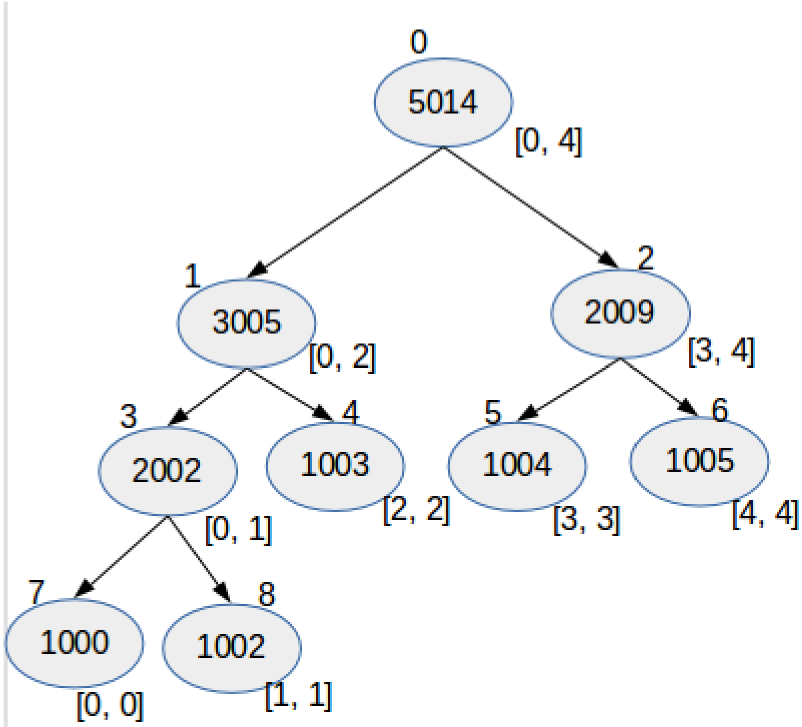
4
1005

0
1000

1
1002

- Les feuilles de l’arbre sont les éléments du tableau.
- Chaque niveau représente la plage des entiers à sommes.

Représentation conceptuelle du segment tree



Représentation algorithmique du segment tree

segTree								
0	1	2	3	4	5	6	7	8
5014	3005	2009	2002	1003	5004	1005	1000	2002

Calcul de la taille du segment tree

Si la taille (*len*) du tableau d'origine est une puissance de 2 alors la taille du segment tree est égale à *len * 2 – 1*, sinon on cherche la puissance de 2 directement supérieur à *len* et on applique la même formule.

On peut trouver la taille du segment tree par la formule :

$2*2^{\lceil \log_2 N \rceil}-1$, N est la taille du tableau d'origine.

Fils gauche, fils droit et parent dans un segment tree

- Le fils gauche d'un nœud i est à la position 2 * i + 1
- Le fils droit d'un nœud i est à la position 2 * i + 2
- Le parent d'un nœud i est à la position (i – 1) / 2

Fonctionnement d’un segment tree

Revenons à notre exemple :

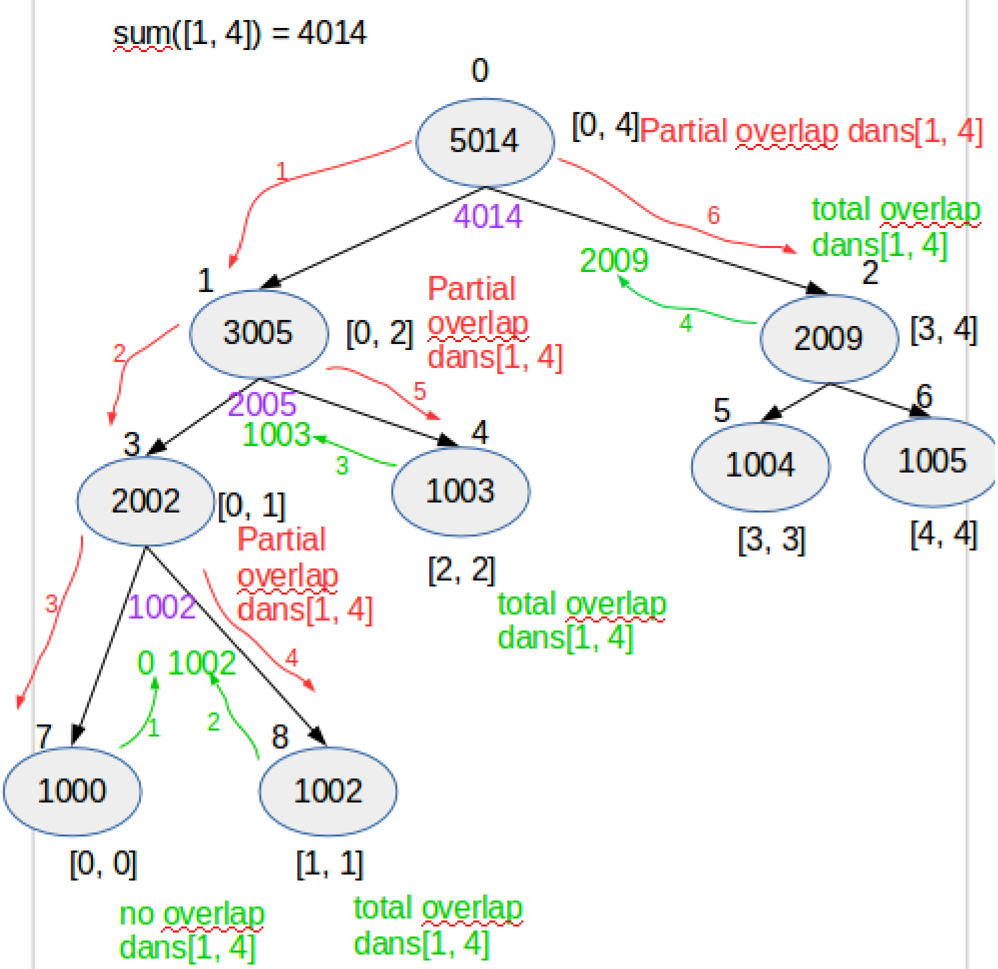
input				
0	1	2	3	4
1000	1002	1003	1004	1005

Cherchons la somme de [1, 4] = ?

Avec le tableau, sum([1, 4]) = 4014.

Avec le segment tree, on commence par le nœud 0 (la racine) de l’arbre et on détermine si :

- L’intervalle du nœud est partiellement imbriqué dans l’intervalle de la requête (partial overlap), si oui, on descend vers le fils gauche et le fils droit.
- L’intervalle est totalement imbriqué dans l’intervalle de la requête (total overlap), si oui, on retourne la valeur du nœud courant.
- L’intervalle n’est pas imbriqué dans l’intervalle de la requête, si oui, on retourne 0.



À la racine, on a un partial overlap parce que [0, 4] n’est pas totalement imbriqué dans [1, 4], donc On descend vers le fils gauche de la racine.

Le nœud 1 → partial overlap ⇒ on descend vers le nœud 3.

Le nœud 3 → partial overlap ⇒ on descend vers le nœud 7.

(A) : Le nœud 7 → no overlap ⇒ on retourne 0.

on revient au nœud 3 et on descend vers le nœud 8.

(B) : Le nœud 8 → total overlap ⇒ on retourne 1002.

(A)+(B) = (C) : La somme de 0 et 1002 = 1002

On revient au nœud 1 est on descend vers le nœud 4.

(D) : Le nœud 4 → total overlap ⇒ on retourne 1003.

(C)+(D) = (E) : La somme de 1002 et 1003 = 2005

On revient au nœud 0 est on descend vers le nœud 2.

(F) : Le nœud 2 → total overlap ⇒ on retourne 2009.

(E)+(F) : La somme de 2005 et 2009 = 4014

Implémentation de la fonction de construction du segment tree

Déclaration des tableaux

```
typedef long long ll;
ll *input = NULL, *segTree = NULL;
```

Fonction buildSegTree : O(N) //N est la taille du sgement tree

```
//low : l’indice 0 de tableau d’origine
//high : taille du tableau d’origine – 1
//pos : l’indice du root (0)
void buildSegtTree(ll low, ll high, ll pos){
  if (low == high) {
    segTree[pos] = input[low];
    return;
  }
  ll mid = (high + low) / 2;
  buildSegtTree(low, mid, 2*pos+1);          1
  buildSegtTree(mid+1, high, 2*pos+2);         2
  segTree[pos] = segTree[2*pos+1] + segTree[2*pos+2];
}
```

Exécution à la main

input				
0	1	2	3	4
1000	1002	1003	1004	1005

segTree														
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

L'appel dans le main : buildSegtTree(0, N-1, 0) ;

low	high	pos	mid	Num. ligne
0	4	0	2	1
0	2	1	1	1
0	1	3	0	1
0	0	7	return	

segTree														
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	0	0	0	0	0	0	1000	0	0	0	0	0	0	0

low	high	pos	mid	Num. ligne
0	4	0	2	1
0	2	1	1	1
0	1	3	0	2
1	1	8	return	

segTree														
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	0	0	0	0	0	0	1000	1002	0	0	0	0	0	0

low	high	pos	mid	Num. ligne
0	4	0	2	1
0	2	1	1	1
0	1	3	0	2
segTree[3] = segTree[7] + segTree[8];				

segTree														
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	0	0	2002	0	0	0	1000	1002	0	0	0	0	0	0

low	high	pos	mid	Num. ligne
0	4	0	2	1
0	2	1	1	2
2	2	4	return	

segTree														
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	0	0	2002	1003	0	0	1000	1002	0	0	0	0	0	0

low	high	pos	mid	Num. ligne
0	4	0	2	1
0	2	1	1	2
segTree[1] = segTree[3] + segTree[4];				

segTree														
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	3005	0	2002	1003	0	0	1000	1002	0	0	0	0	0	0

low	high	pos	mid	Num. ligne
0	4	0	2	2
3	4	2	3	1
3	3	5	return	

segTree														
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	3005	0	2002	1003	1004	0	1000	1002	0	0	0	0	0	0

<table>
<tr><th colspan="3">low</th><th colspan="3">high</th><th colspan="3">pos</th><th colspan="3">mid</th><th colspan="3">Num. ligne</th></tr>
<tr><td colspan="3">0</td><td colspan="3">4</td><td colspan="3">0</td><td colspan="3">2</td><td colspan="3">2</td></tr>
<tr><td colspan="3">3</td><td colspan="3">4</td><td colspan="3">2</td><td colspan="3">3</td><td colspan="3">2</td></tr>
<tr><td colspan="3">4</td><td colspan="3">4</td><td colspan="3">6</td><td colspan="6">return</td></tr>
<tr><th colspan="15">segTree</th></tr>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td></tr>
<tr><td>0</td><td>3005</td><td>0</td><td>2002</td><td>1003</td><td>1004</td><td>1005</td><td>1000</td><td>1002</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
</table>

<table>
<tr><th colspan="3">low</th><th colspan="3">high</th><th colspan="3">pos</th><th colspan="3">mid</th><th colspan="3">Num. ligne</th></tr>
<tr><td colspan="3">0</td><td colspan="3">4</td><td colspan="3">0</td><td colspan="3">2</td><td colspan="3">2</td></tr>
<tr><td colspan="3">3</td><td colspan="3">4</td><td colspan="3">2</td><td colspan="3">3</td><td colspan="3">2</td></tr>
<tr><td colspan="15">segTree[2] = segTree[5] + segTree[6];</td></tr>
<tr><th colspan="15">segTree</th></tr>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td></tr>
<tr><td>0</td><td>3005</td><td>2009</td><td>2002</td><td>1003</td><td>1004</td><td>1005</td><td>1000</td><td>1002</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
</table>

<table>
<tr><th colspan="3">low</th><th colspan="3">high</th><th colspan="3">pos</th><th colspan="3">mid</th><th colspan="3">Num. ligne</th></tr>
<tr><td colspan="3">0</td><td colspan="3">4</td><td colspan="3">0</td><td colspan="3">2</td><td colspan="3">2</td></tr>
<tr><td colspan="15">segTree[0] = segTree[1] + segTree[2];</td></tr>
<tr><th colspan="15">segTree</th></tr>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td></tr>
<tr><td>5014</td><td>3005</td><td>2009</td><td>2002</td><td>1003</td><td>1004</td><td>1005</td><td>1000</td><td>1002</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
</table>

Implémentation de la fonction query (range query)

Fonction query : O(log N) //N est la taille du sgement tree

```
//low : l’indice 0 de tableau d’origine
//high : taille du tableau d’origine – 1
//pos : l’indice du root (0)
//qlow : début de la plage
::qhigh : fin de la plage
ll query(ll qlow, ll qhigh, ll low, ll high, ll pos) {
  //Total overlap
  if (qlow <= low && qhigh >= high)
    return segTree[pos];

  //No overlap
  if (qlow > high || qhigh < low)
    return 0;

  //Partial overlap
  ll mid = low + (high - low) / 2;
  return query(qlow, qhigh, low, mid, 2*pos+1) +              1
         query(qlow, qhigh, mid+1, high, 2*pos+2);          2
}
```

Exécution à la main

segTree														
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
5014	3005	2009	2002	1003	1004	1005	1000	1002	0	0	0	0	0	0

sum ([1, 4]) = ?

qlow	qhigh	low	high	pos	mid	Num. ligne
1	4	0	4	0	2	1
1	4	0	2	1	1	1
1	4	0	1	3	0	1
1	4	0	0	7	No overlap return 0	

qlow	qhigh	low	high	pos	mid	Num. ligne
1	4	0	4	0	2	1
1	4	0	2	1	1	1
1	4	0	1	3	0	2
1	4	1	1	8	Total overlap return segTree[8] = 1002	

qlow	qhigh	low	high	pos	mid	Num. ligne
1	4	0	4	0	2	1
1	4	0	2	1	1	1
1	4	0	1	3	0	2
Return 0 + 1002 = 1002						

qlow	qhigh	low	high	pos	mid	Num. ligne
1	4	0	4	0	2	1
1	4	0	2	1	1	2
1	4	2	2	4	Total overlap return segTree[4] = 1003	

qlow	qhigh	low	high	pos	mid	Num. ligne
1	4	0	4	0	2	1
1	4	0	2	1	1	2
Return 1002 + 1003 = 2005						

qlow	qhigh	low	high	pos	mid	Num. ligne
1	4	0	4	0	2	2
1	4	3	4	2	Total overlap return segTree[2] = 2009	
Return 2005 + 2009 = 4014						

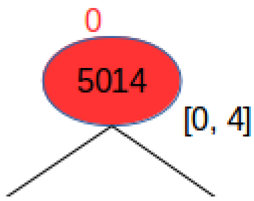
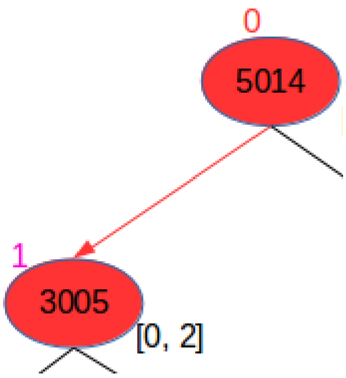
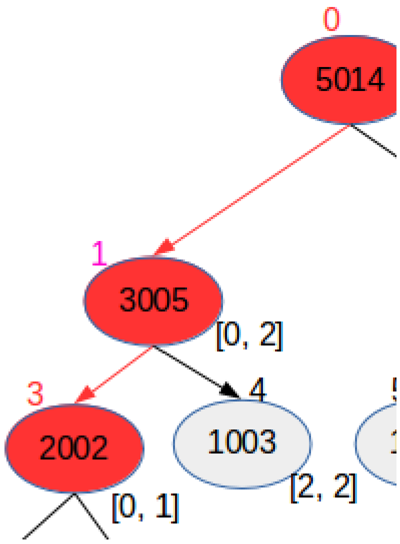
Implémentation de la fonction update (point update)

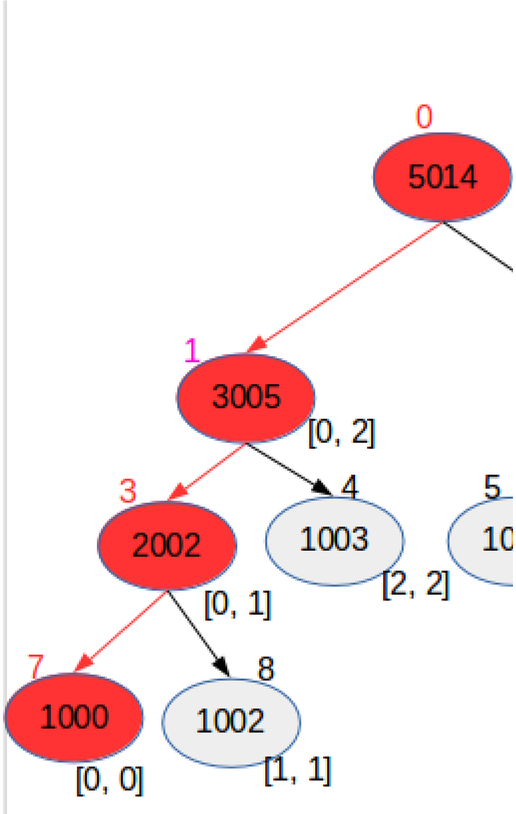
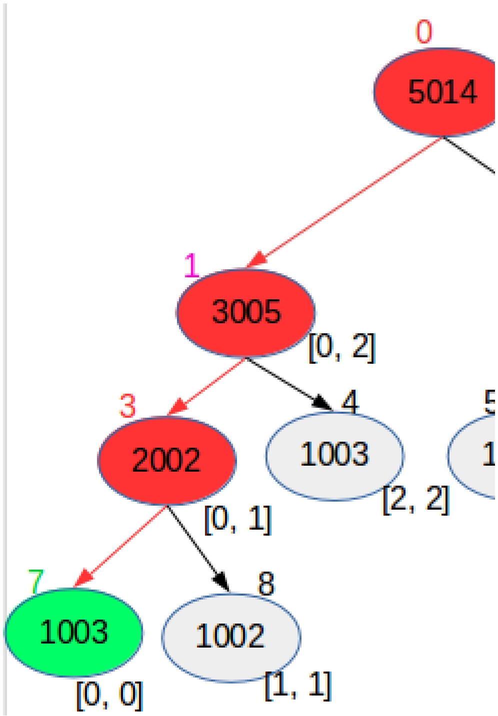
La mise à jour d’un élément du tableau original à un certain indice, nécessite la reconstruction de tout l’arbre. Parce qu’il y a des nœuds qui font la somme du nœud modifié.

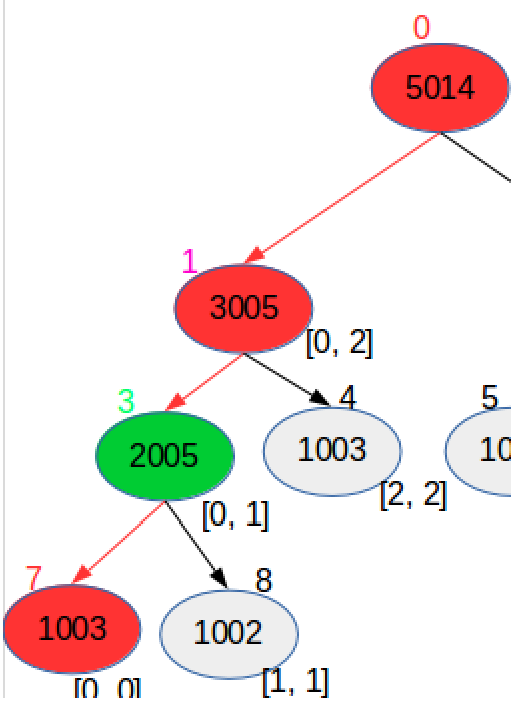
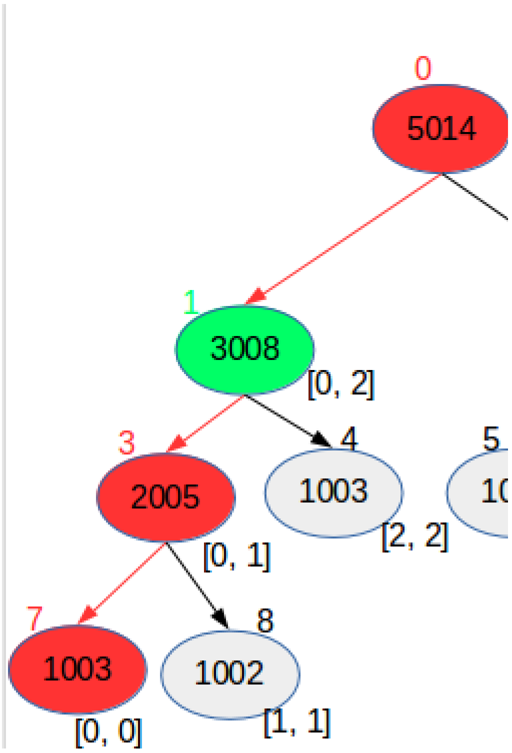
On commence par la racine du segment tree, si le milieu du tableau d’origine (input) est supérieur ou égale à l’indice de l’élément à modifier dans le tableau d’origine, alors on descend vers le fils gauche de l’arbre. Sinon, on descend vers le fils droit. On s’arrête que lorsque on a arrive à une feuille, à ce moment on met à jour cette dernière et on revient sur les nœuds visités pour une éventuelles mise à jour aussi.

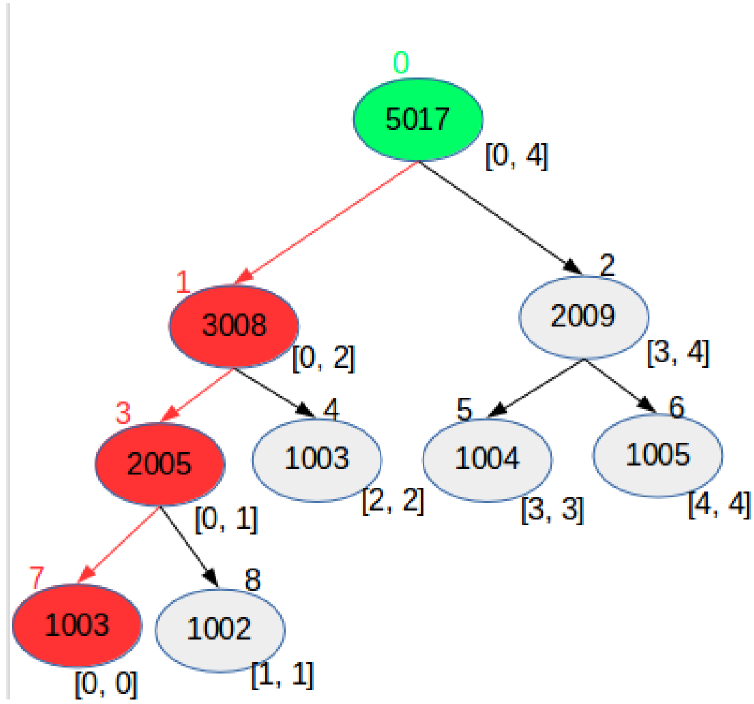
G 0 3

la nouvelle valeur de input[0] est 1003.

On commence par le root :	
le milieu du tableau = 2, on descend vers le fils gauche :	
le milieu du tableau = 1, on descend vers le fils gauche :	

le milieu du tableau = 0, on descend vers le fils gauche :	
Le nœud 7 est une feuille, on le met à jour	

Mise à jour du nœud 3	
Mise à jour du nœud 1	

Mise à jour du nœud 0 (root)	

Fonction update: O(log N) //N est la taille du sgement tree

```
//idx : indice de l’élément à modifier dans le tableau d’origine
//v : la nouvelle valeur de input[idx]
//low : l’indice 0 de tableau d’origine
//high : taille du tableau d’origine – 1
//pos : l’indice du root (0)
void update(ll idx, ll v, ll low, ll high, ll pos){
  //Une feuille
  if (low == high) {
    segTree[pos] = v;
    return;
  }

  ll mid = low + (high - low) / 2;

  if (idx <= mid)
    update(idx, v, low, mid,  2*pos+1);        1
  else if (idx > mid)
    update(idx, v, mid+1, high, 2*pos+2);      2

  segTree[pos] = segTree[2 * pos + 1] + segTree[2 * pos + 2];
}
```

Exécution à la main

G 0 3

idx	v	low	high	pos	mid	Num. ligne
0	1003	0	4	0	2	1
0	1003	0	2	1	1	1
0	1003	0	1	3	0	1
0	1003	0	0	7	0	

segTree

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
5014	3005	2009	2002	1003	1004	1005	1003	1002	0	0	0	0	0	0

idx	v	low	high	pos	mid	Num. ligne
0	1003	0	4	0	2	1
0	1003	0	2	1	1	1
0	1003	0	1	3	0	1

segTree[3] = segTree[7] + segTree[8];

segTree

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
5014	3005	2009	2005	1003	1004	1005	1003	1002	0	0	0	0	0	0

idx	v	low	high	pos	mid	Num. ligne
0	1003	0	4	0	2	1
0	1003	0	2	1	1	1

segTree[1] = segTree[3] + segTree[4];

segTree

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
5014	3008	2009	2005	1003	1004	1005	1003	1002	0	0	0	0	0	0

idx	v	low	high	pos	mid	Num. ligne
0	1003	0	4	0	2	1
segTree[0] = segTree[1] + segTree[2];						

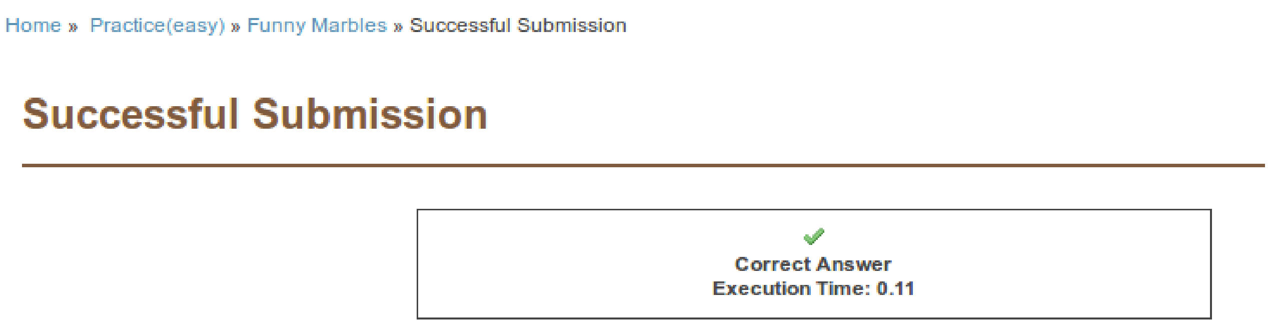
segTree														
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
5017	3008	2009	2005	1003	1004	1005	1003	1002	0	0	0	0	0	0

Code complet : O(Q log SN)

Submission accepted : https://www.codechef.com/viewsolution/13043669
ideone : http://ideone.com/Bt4P9v

```
#include <cstdio>
#include <cstdlib>
#include <cmath>

typedef long long ll;

ll *input = NULL, *segTree = NULL;
ll N, Q;

void buildSegtTree(ll low, ll high, ll pos){
  if (low == high) {
    segTree[pos] = input[low];
    return;
  }
  ll mid = (high + low) / 2;
  buildSegtTree(low, mid, 2*pos+1);
  buildSegtTree(mid+1, high, 2*pos+2);
  segTree[pos] = segTree[2*pos+1] + segTree[2*pos+2];
}


ll query(ll qlow, ll qhigh, ll low, ll high, ll pos) {
  if (qlow <= low && qhigh >= high)
    return segTree[pos];
  if (qlow > high || qhigh < low)
    return 0;
  ll mid = low + (high - low) / 2;
  return query(qlow, qhigh, low, mid, 2*pos+1) +
         query(qlow, qhigh, mid+1, high, 2*pos+2);
}


void update(ll idx, ll v, ll low, ll high, ll pos){
  if (low == high) {
    segTree[pos] = v;
    return;
  }

  ll mid = low + (high - low) / 2;

  if (idx > mid)
    update(idx, v, mid+1, high, 2*pos+2);
  else if (idx <= mid)
    update(idx, v, low, mid,  2*pos+1);

  segTree[pos] = segTree[2 * pos + 1] + segTree[2 * pos + 2];
}
int main() {
  scanf("%lld%lld", &N, &Q);

  input = new ll [N];
```

```
  for (ll i = 0 ; i < N ; ++i) {
    ll ai;
    scanf("%lld", &ai );
    input[i] = ai;
  }


  ll SN = (ll)(ceil(log2(N)));
  //SN = 2*(ll)pow(2, SN) - 1;
  SN = 2 * (1 << SN) - 1;
  segTree = new ll[SN];
  buildSegtTree(0, N-1, 0);


  for (ll i = 0 ; i < Q ; ++i) {
    char q;
    ll l, r;
    scanf(" %c", &q);
    if (q == 'S') {
      ll l, r;
      scanf("%lld%lld",&l, &r );
      printf("%lld\n", query(l, r, 0, N-1, 0));
    }
    if (q == 'G') {
      ll idx, v;
      scanf("%lld%lld",&idx, &v );
      v += input[idx];
      update(idx, v, 0, N-1, 0);
    }
    if (q == 'T') {
      ll idx, v;
      scanf("%lld%lld",&idx, &v );
      v -= input[idx];
      update(idx, -v, 0, N-1, 0);
    }
  }
  return 0;
}
```



Range update & range query

Problème :
http://www.spoj.com/problems/HORRIBLE/
Le problème demande que la mise à jour du tableau soit de l'indice *p* à *q*. Une solution est de mettre à jour les éléments un par un. La complexité temporelle de cette approche est égale à O(N log N).

Cette solution pose d'autres problème, autre que l'augmentation de temps d'exécution.

Comme nous avons déjà vu, la mise à jour d'un élément du tableau d'origine, nécessite la reconstruction du segment tree, en commençant par les feuilles et en se terminant par la racine, en passant par touts les nœuds parents. D'où le nom du problème *ancestral locality*. Donc une nœud peut être mise à jour plusieurs fois. Et il se peut que la mise à jour est effectuée sur un ensemble du nœuds qui ne sont pas demandés lors de la requête. Par exemple, si la requête de mise à jour demande l'ajout d'une valeur à tout les éléments du tableau de l'indice 6 à 9, et que la requête demande la somme de 0 à 3. Au niveau du segment tree, la racine par exemple, est mis à jour 4 fois, or que en réalité elle n'était pas concernée par la mise à jour.

Utiliser une propagation fainéante (lazy propagation) résout tout ces problèmes. Comme son nom l'indique, la mise à jour (ou une requête) ne concerne que les nœuds en question.

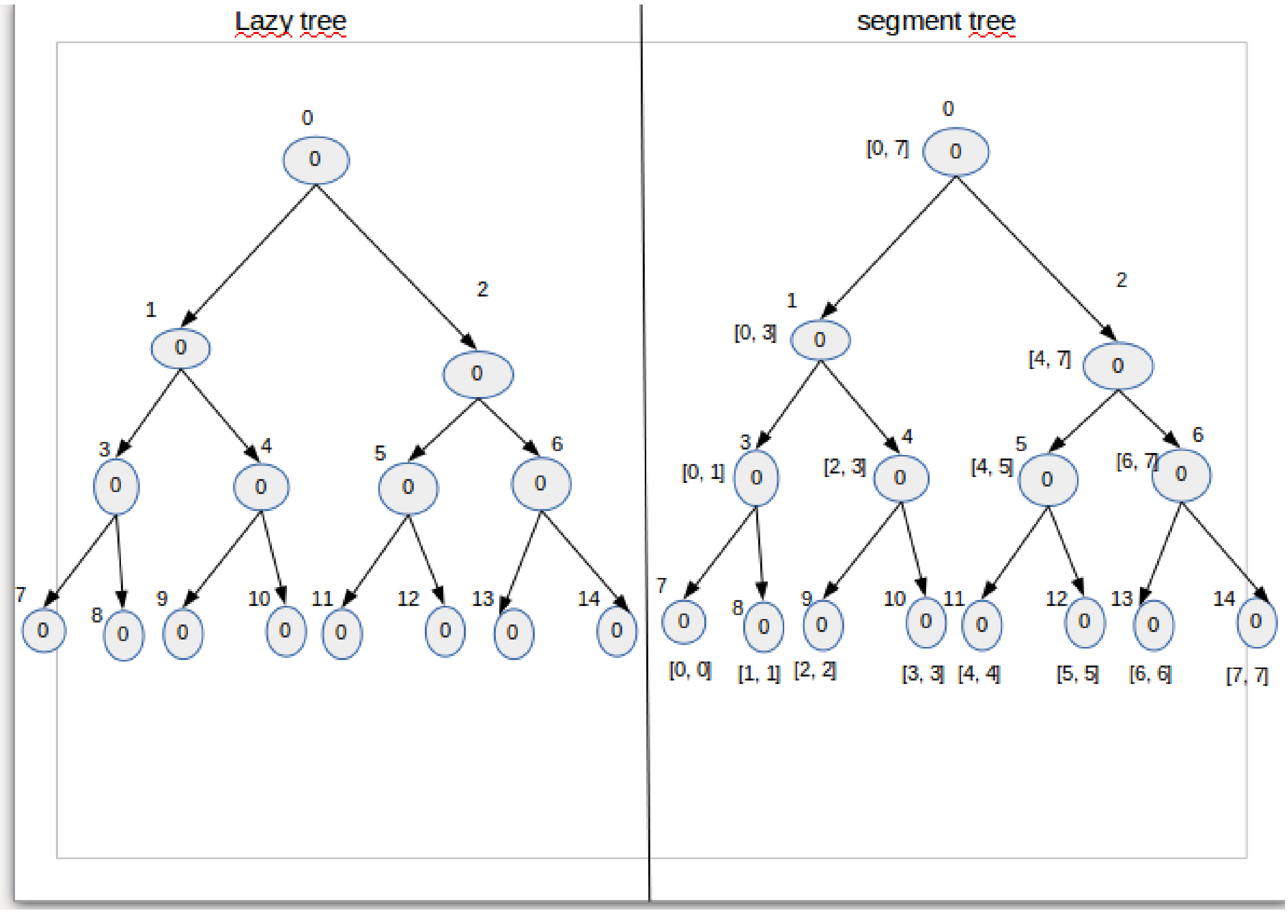
Durant une requête de sommation ou une autre requête de mise à jour, il faut être sur que tout les propagations sont réalisé aux nœuds concernés par ces requêtes. En bref, on a va reporter la mise à jour des nœuds fils jusqu'à on demande l'accès à ces nœuds.

Pour ce faire, il faut maintenir un autre arbre (lazy tree) qui a la même taille que le segment tree. *lazy[i]* maintient la valeur avec lequel un *nœud i* va être incrémenter en cas de besoin. Si *lazy[i]* est égale à zéro, cela veut dire que un *nœud i* est mis à jour.

Prenons l'exemple du problème :

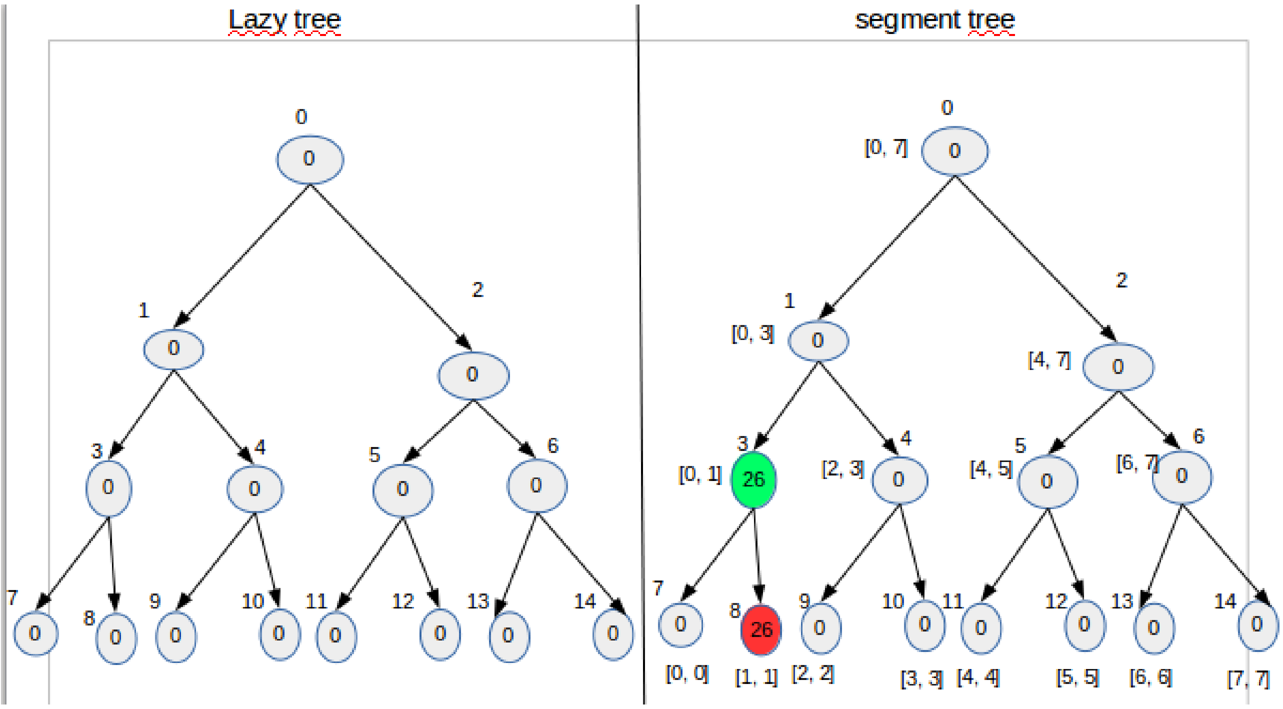
```
0 2 4 26
0 4 8 80
0 4 5 20
1 8 8
0 5 7 14
1 4 8
```

Au début, on a :

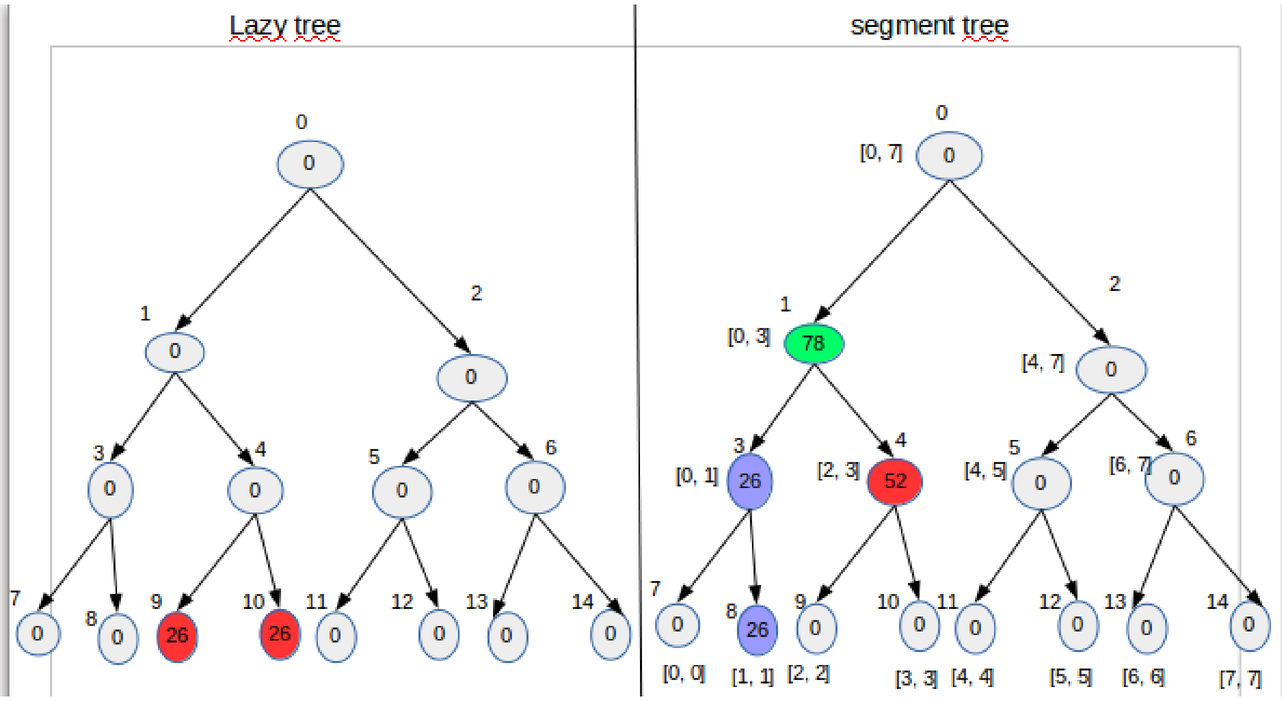


query #1 : 0 2 4 26 => incrémenter 26 de l'indice 1 à 3

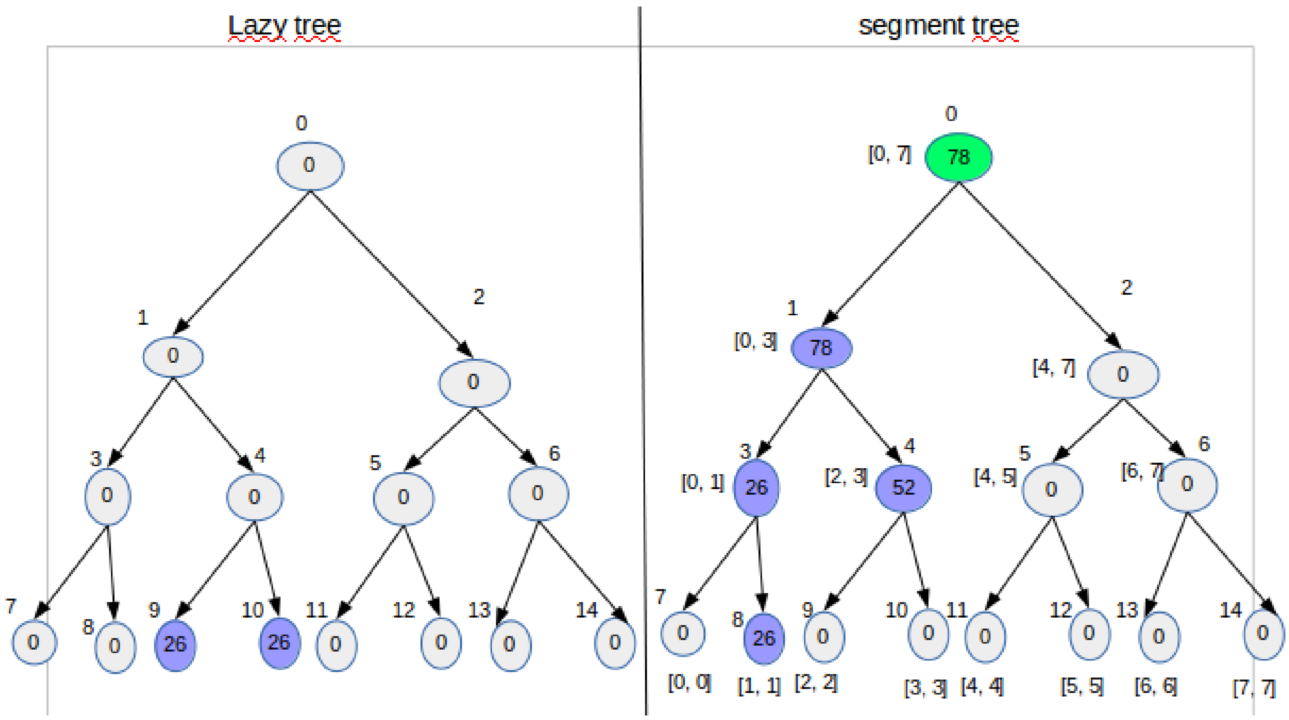
pos	lazy[pos]	Mis à jour ?	Overlap ?	Nœuds suivants
0	0	oui	partial	1 puis 2
1	0	oui	partial	3 et 4
3	0	oui	partail	7 et 8
7	0	oui	no	8
8	0	oui	total	4
le nœud 8 est mis à jour, nous avons un total overlap => on met à jour le nœud 8 et on met à jour le parent de 7 et 8				



pos	lazy[pos]	Mis à jour ?	Overlap ?	Nœuds suivants
4	0	oui	total	2
le nœud 4 est mis à jour, nous avons un total overlap => on met à jour le nœud 4, on marque ces fils à lazy et on met à jour le parent de 3 et 4				

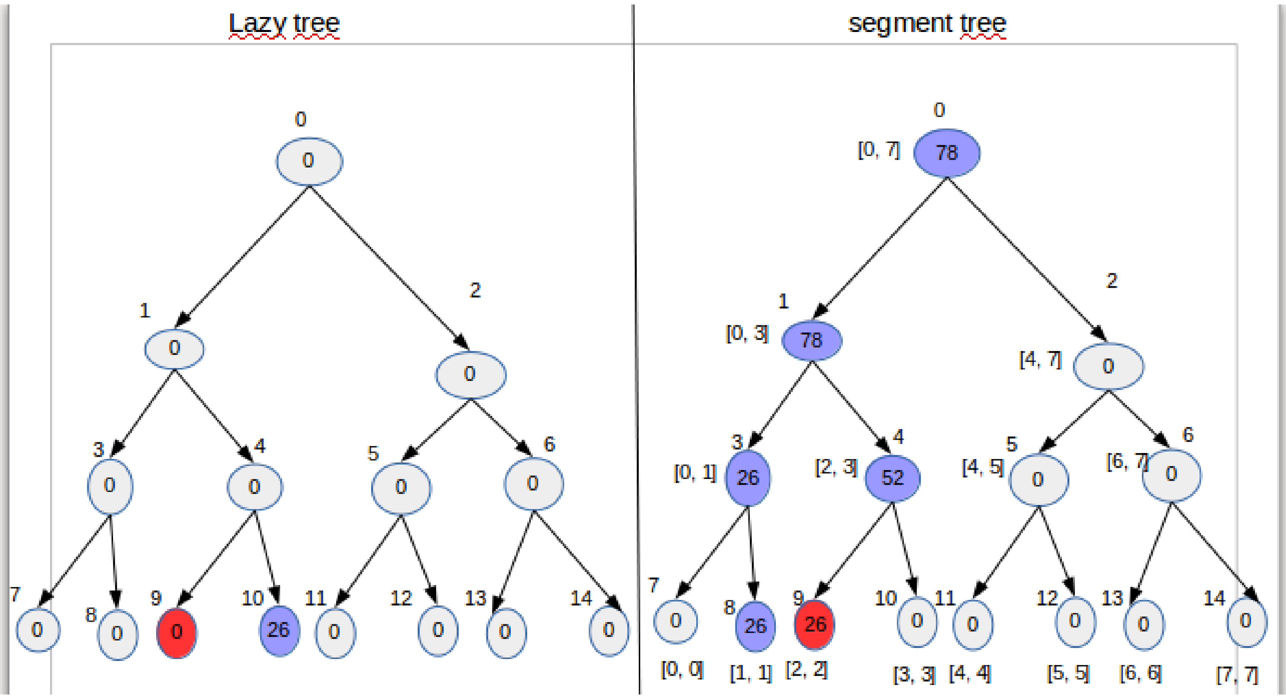


pos	lazy[pos]	Mis à jour ?	Overlap ?	Nœuds suivants
2	0	oui	no	aucun
on met à jour le parent de 1 et 2				

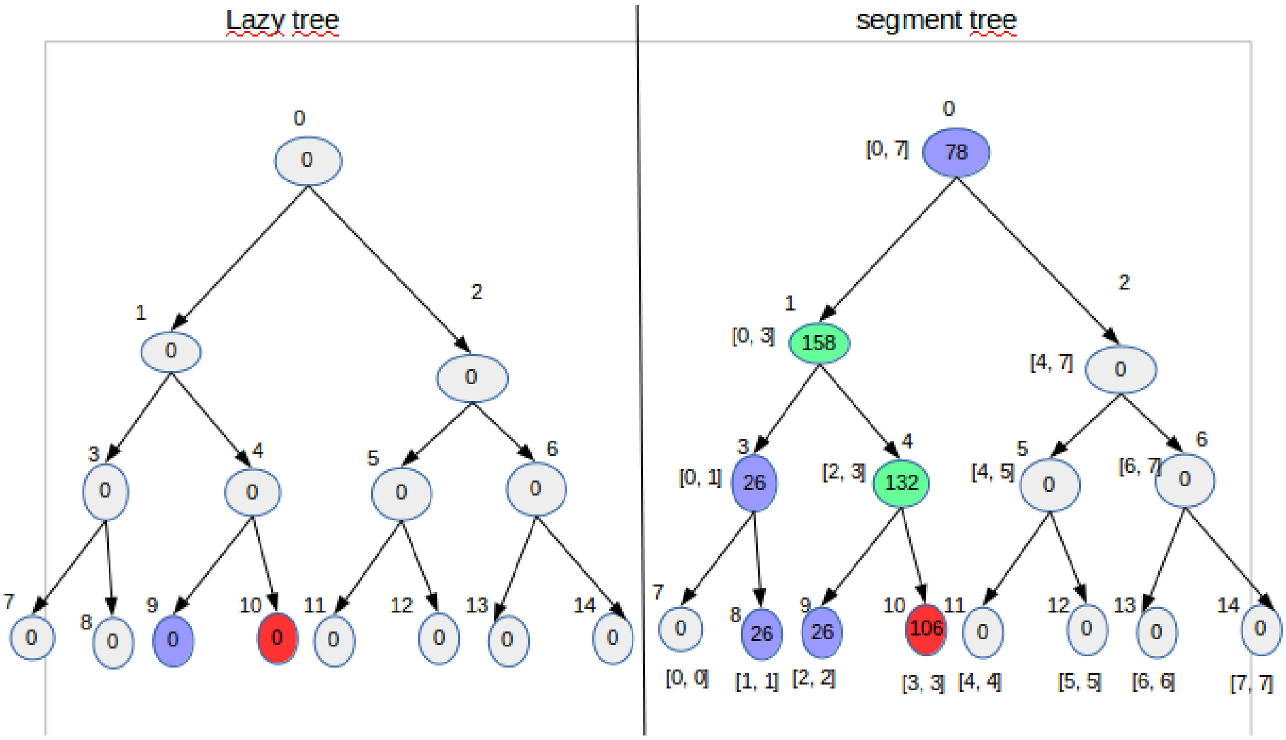


<u>query #2 : 0 4 8 80 => incrémenter 80 de l’indice 3 à 7</u>

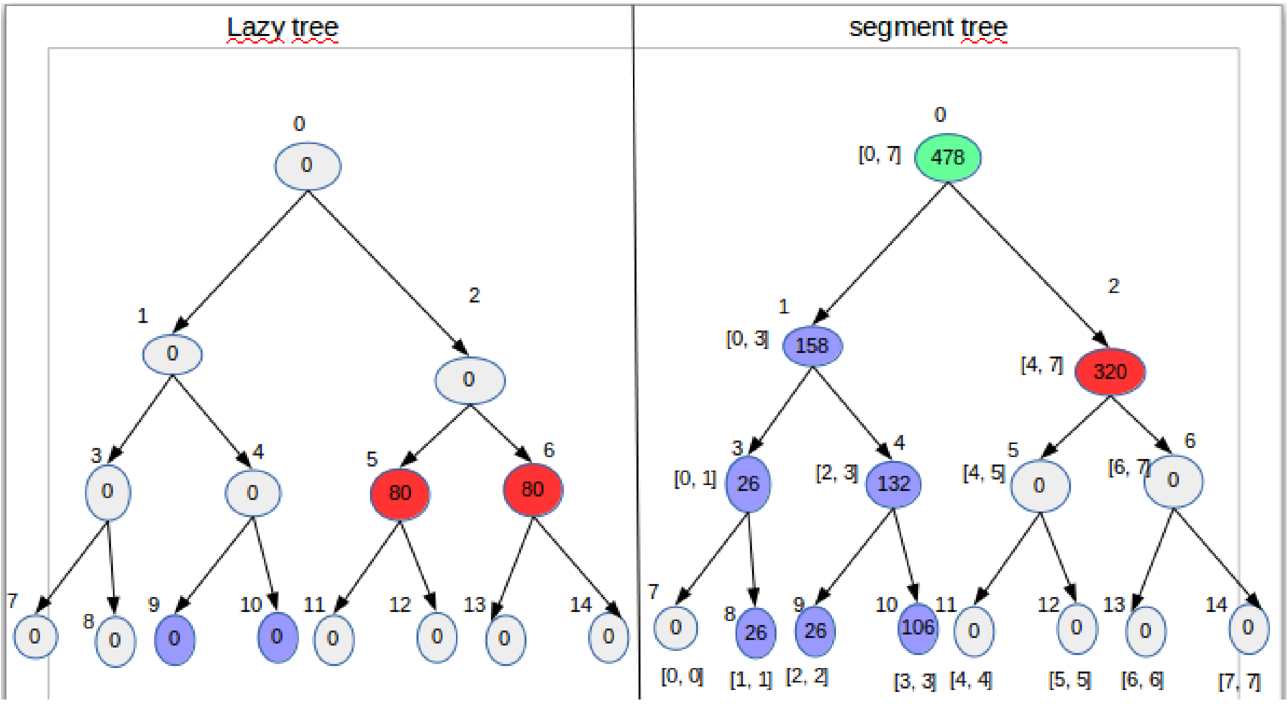
pos	lazy[pos]	Mis à jour ?	Overlap ?	Nœuds suivants
0	0	oui	partial	1 et 2
1	0	oui	partial	3 et 4
3	0	oui	no	4
4	0	oui	partial	9 et 10
9	26	Non (il reste une mise à jour à effectuer)	no	10
Puisque il y a une mise à jour non effectuée au nœud 9, il faut le mettre à jour en ajoutant 26 au nœud 9.				



pos	lazy[pos]	Mis à jour ?	Overlap ?	Nœuds suivants
10	26	Non (il reste une mise à jour à effectuer)	total	2
Puisque il y a une mise à jour non effectuée au nœud 9, il faut le mettre à jour en ajoute 26 + 80 = 106. Et on met à modifie le nœuds parents				

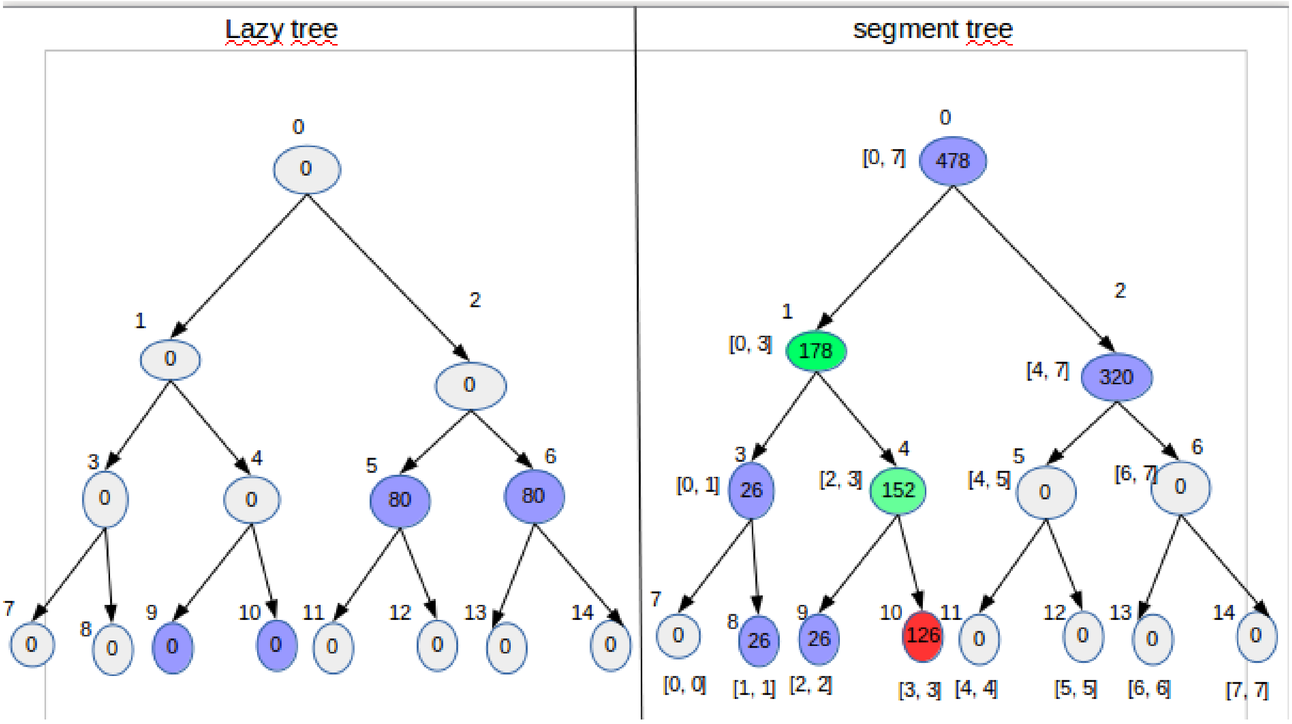


pos	lazy[pos]	Mis à jour ?	Overlap ?	Nœuds suivants
2	0	oui	total	aucun
On met à jour le nœud 2 par sa valeur (320) et on marque ses fils a lazy. Et on modifie le parent de 1 et 2 par la nouvelle somme (320 + 158)				

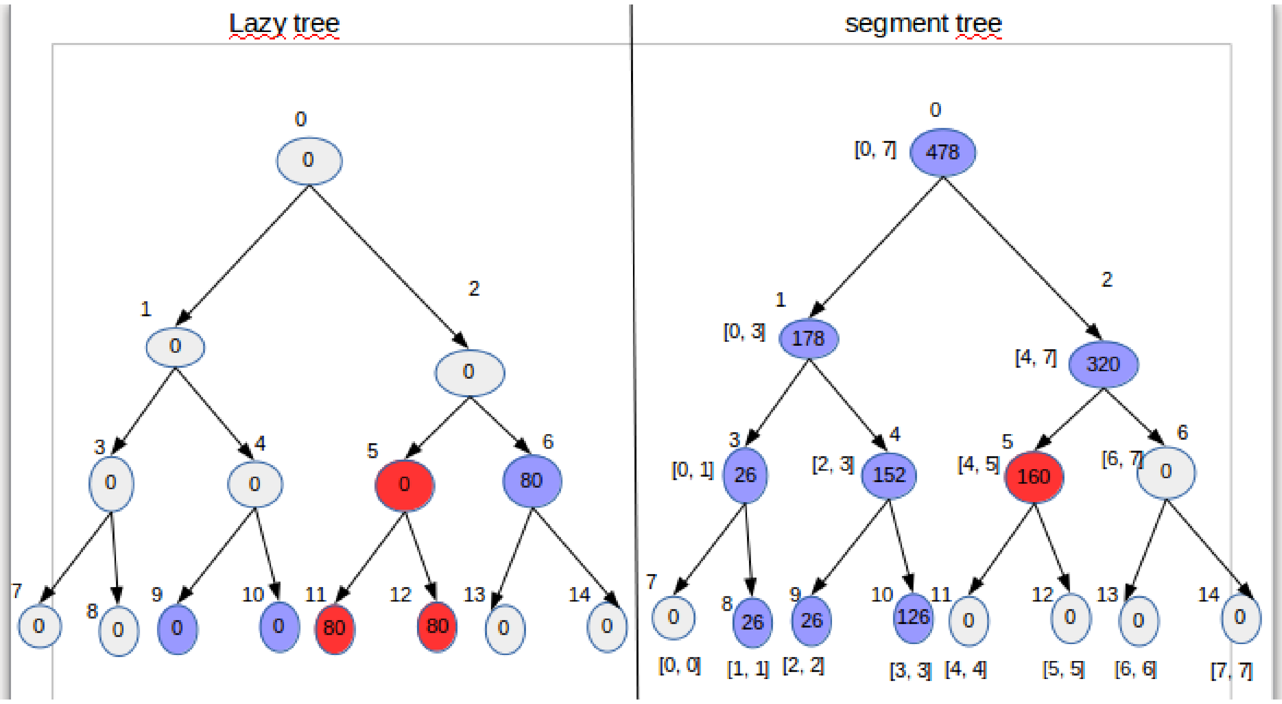


<u>query #3 : 0 4 5 20 => incrémenter 20 de l'indice 3 à 4</u>

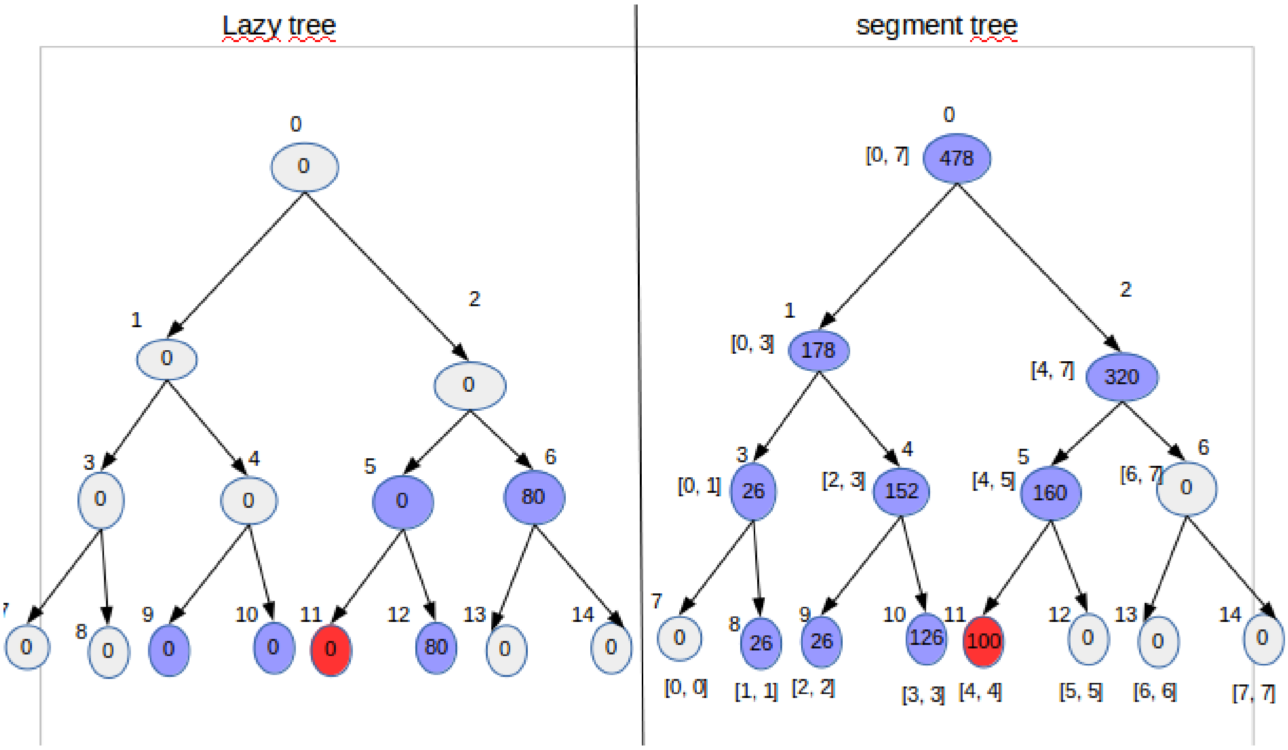
pos	lazy[pos]	Mis à jour ?	Overlap ?	Nœuds suivants
0	0	oui	partial	1 et 2
1	0	oui	partial	3 et 4
3	0	oui	no	4
4	0	oui	partial	9 et 10
9	0	oui	no	10
10	0	oui	total	2
On ajoute 20 au nœud 10 et on met à jour les différents parents.				



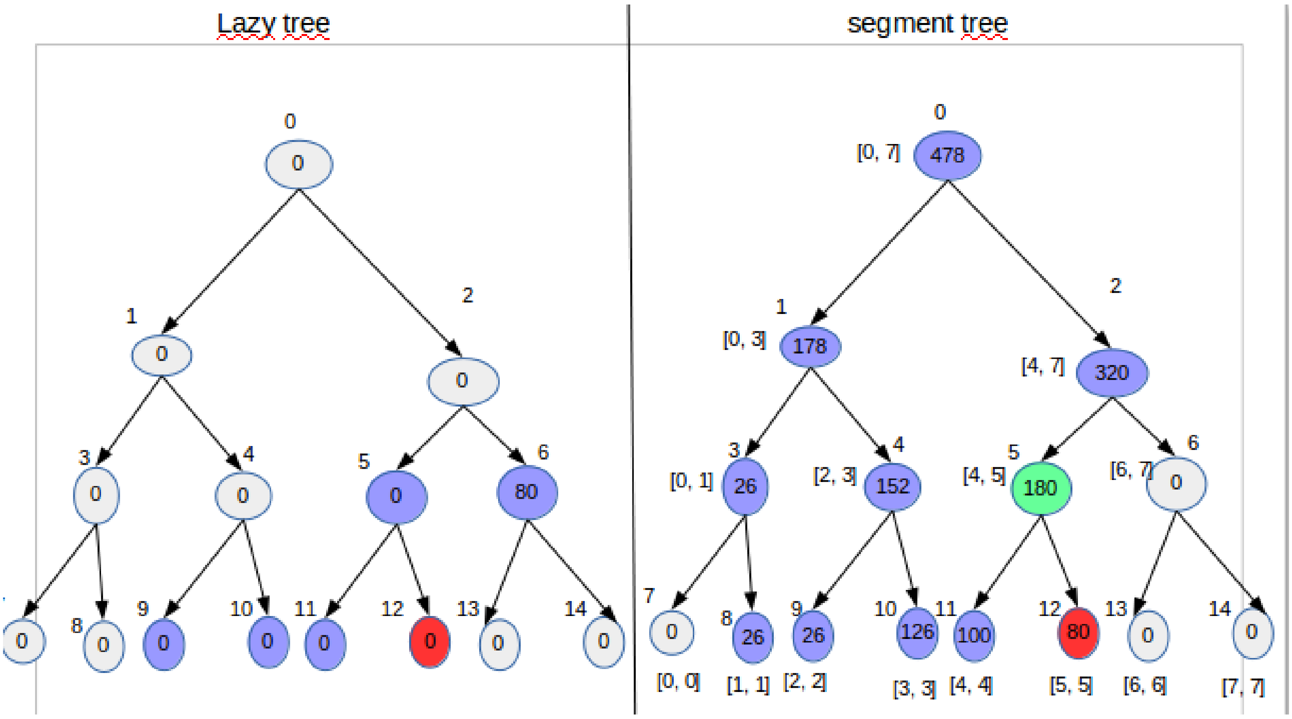
pos	lazy[pos]	Mis à jour ?	Overlap ?	Nœuds suivants
2	0	oui	partial	5 et 6
5	80	non	partial	11 et 12
On ajoute au nœud 5, 2*80 = 160 et on met ses fils a lazy				



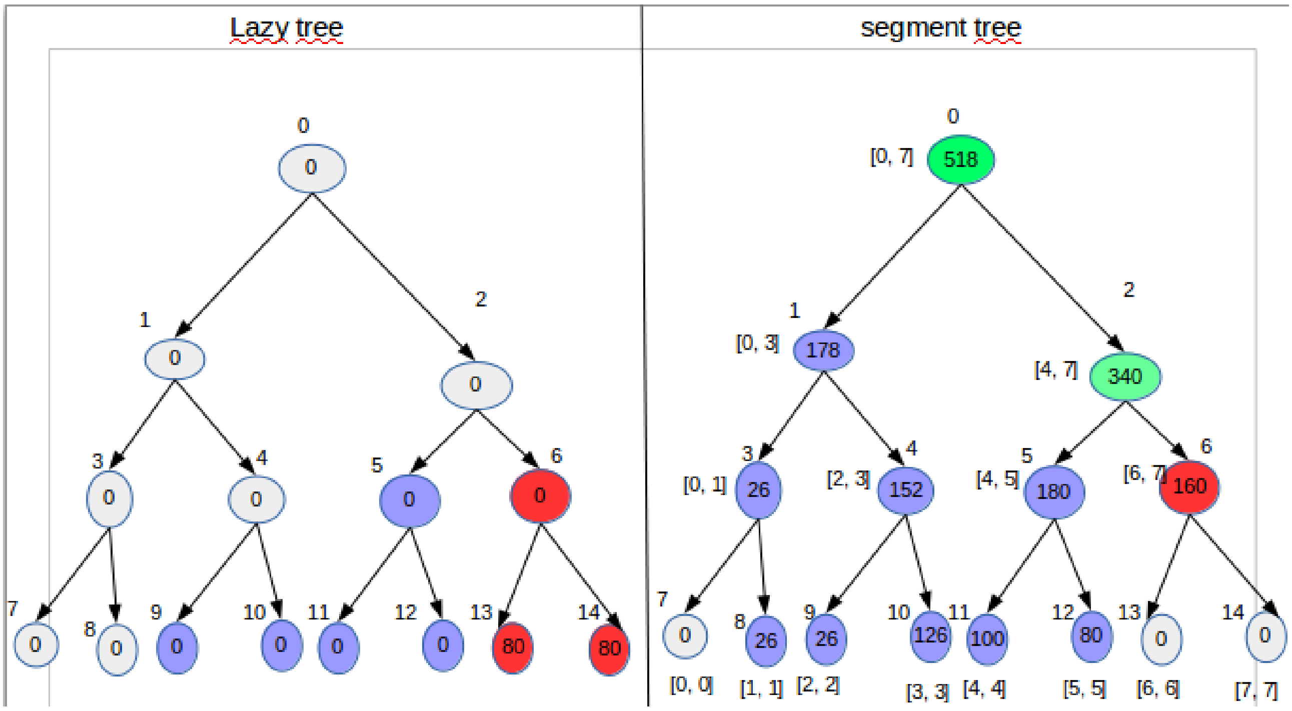
pos	lazy[pos]	Mis à jour ?	Overlap ?	Nœuds suivants
11	80	non	total	12
SegTree[11] = 1 * 80 + 1 * 20 = 100, lazy[11] = 0				



pos	lazy[pos]	Mis à jour ?	Overlap ?	Nœuds suivants
12	80	non	no	6
SegTree[12] = 1 * 80 = 80, lazy[12] = 0				

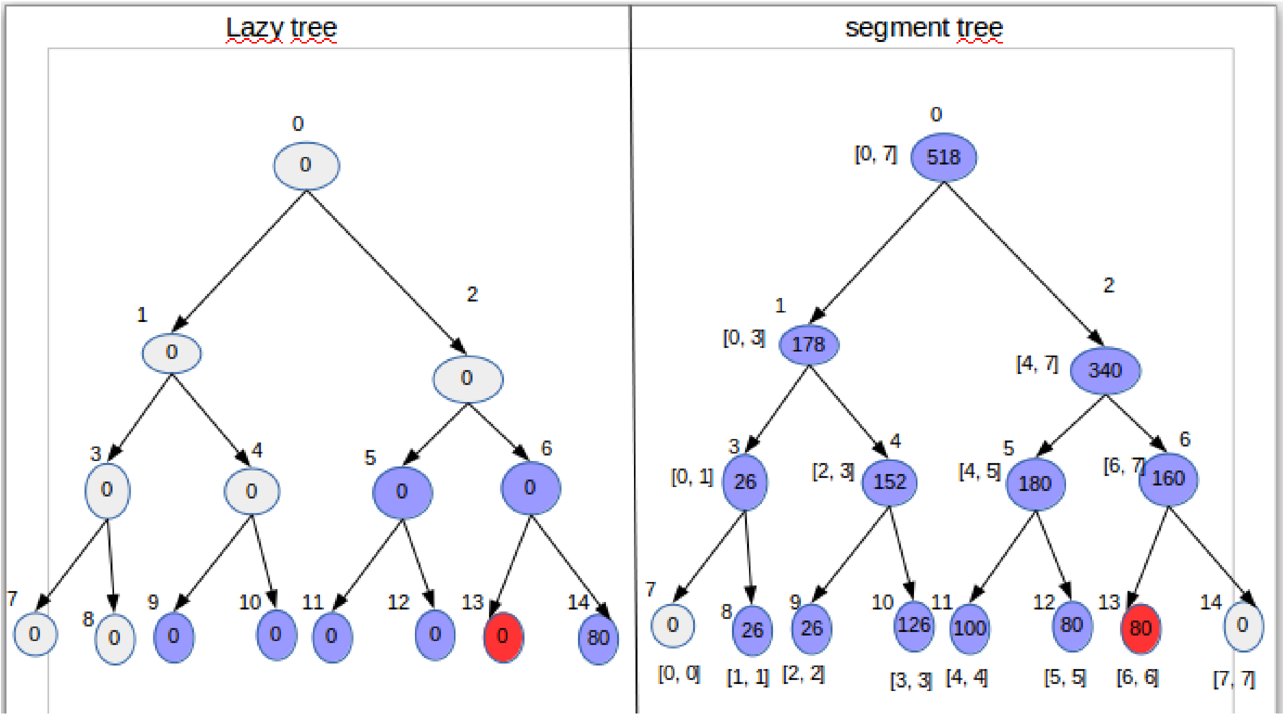


pos	lazy[pos]	Mis à jour ?	Overlap ?	Nœuds suivants
6	80	non	no	aucun
SegTree[6] = 2 * 80 = 160, lazy[6] = 0, lazy[13] = lazy[14] = 80 et on met à jour tout les parents.				



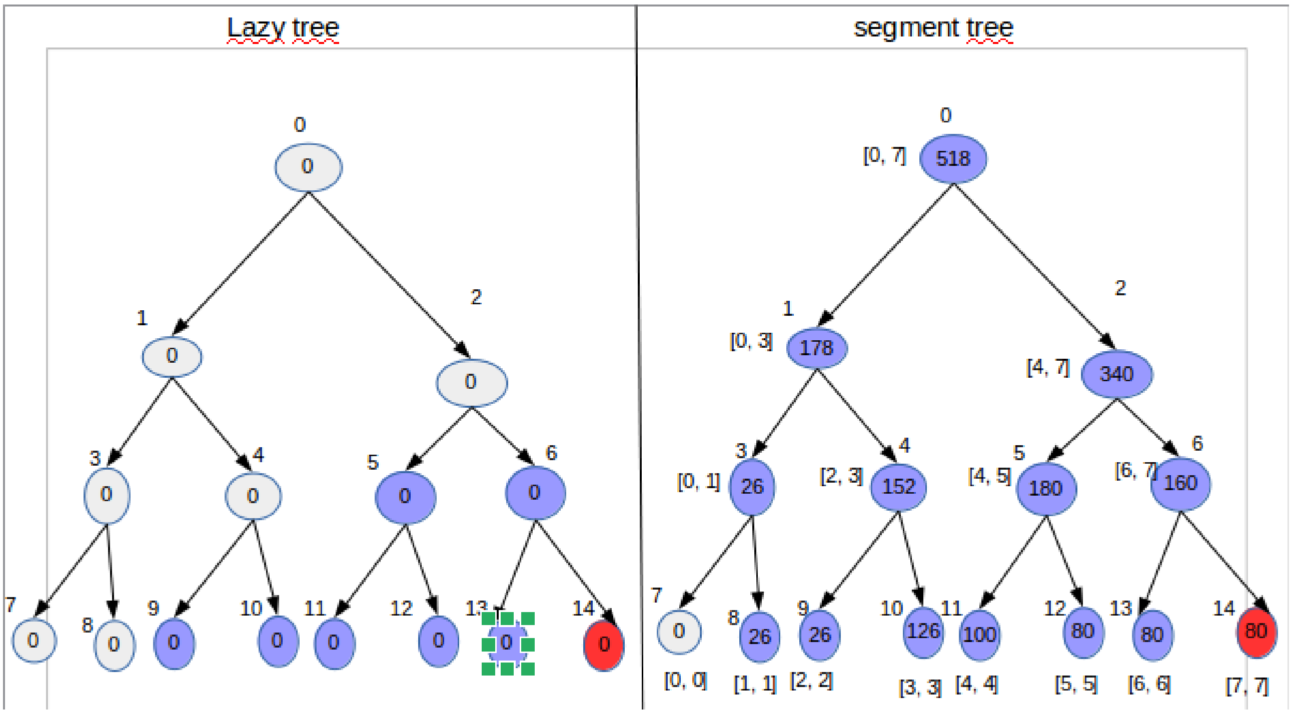
Query #4 : 1 8 8 => sum[7, 7] = ?

pos	lazy[pos]	Mis à jour ?	Overlap ?	Nœuds suivants
0	0	oui	partial	1 et 2
1	0	oui	no	2
2	0	oui	partial	5 et 6
5	0	oui	no	6
6	0	oui	partial	13 et 14
13	80	non	no	14
On met à jour le nœud 13, segTree[13] = 1 * 80 = 80, lazy[13] = 0				



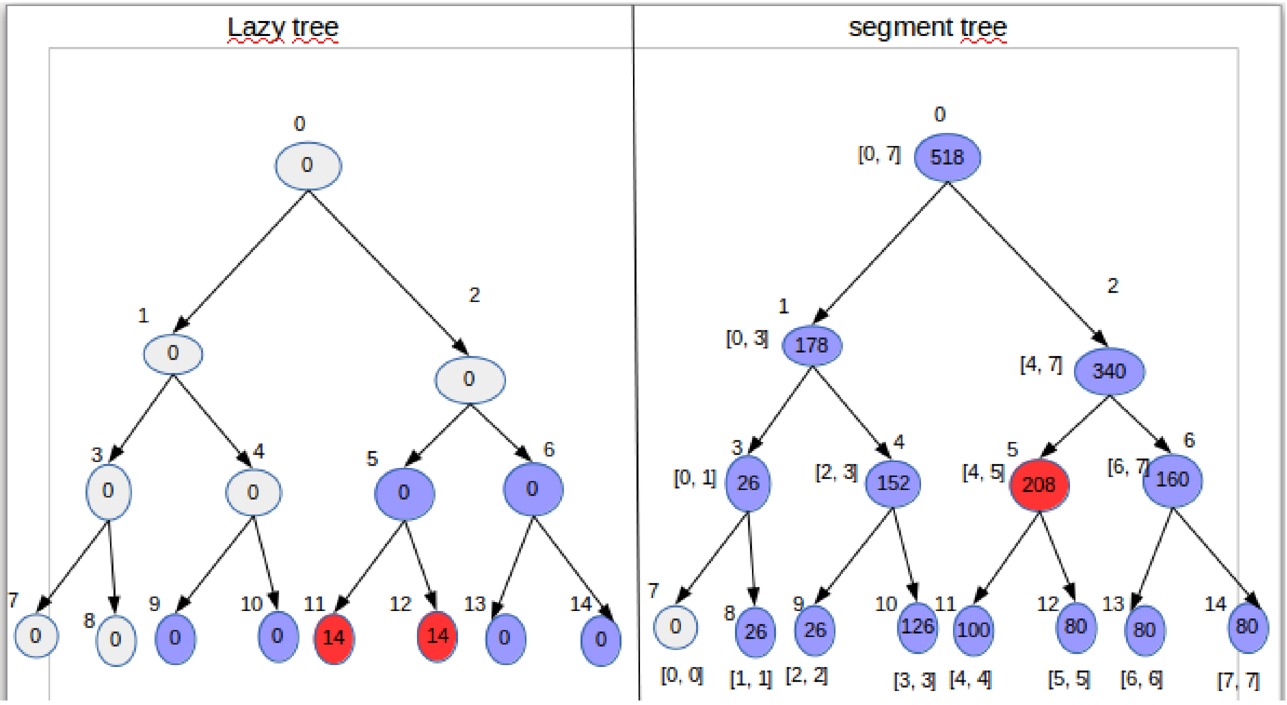
pos	lazy[pos]	Mis à jour ?	Overlap ?	Nœuds suivants
14	80	non	total	aucun
On met à jour le nœud 14, segTree[14] = 1 * 80 = 80, lazy[14] = 0, on retourne 80.				

sum[7, 7] = 80

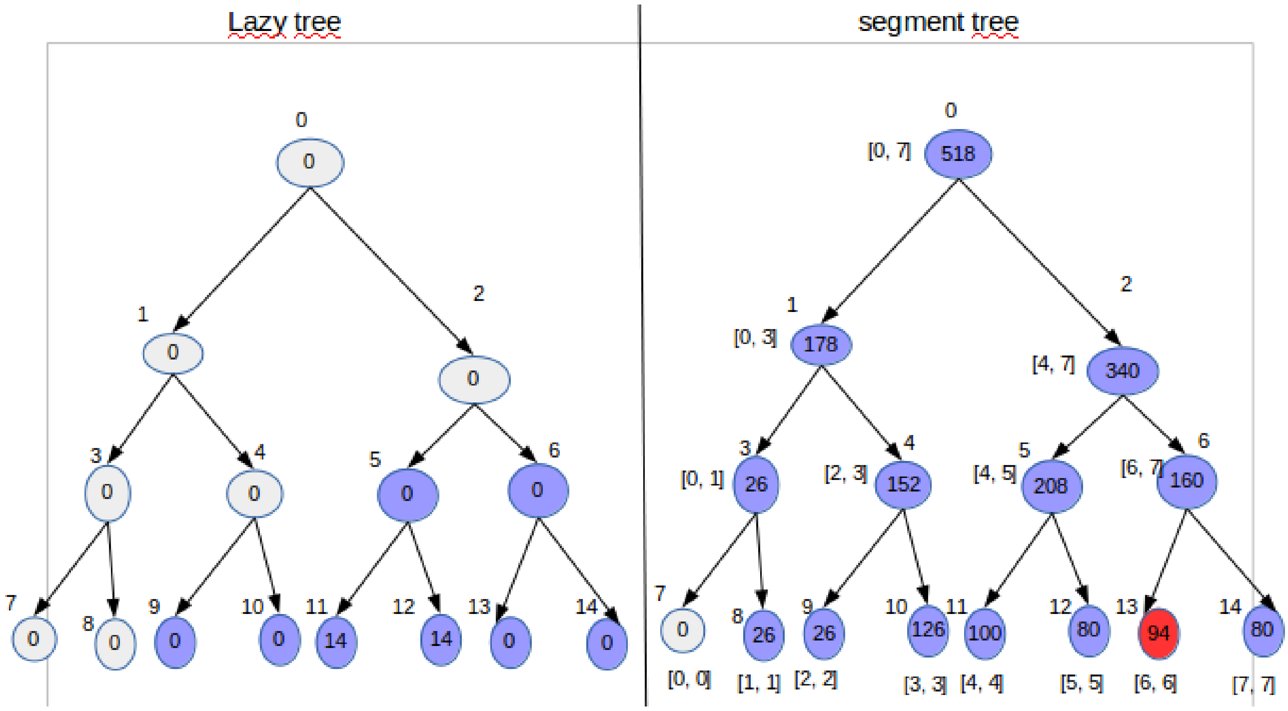


<u>query #5 : 0 5 7 14 => incrémenter 14 de l'indice 4 à 6</u>

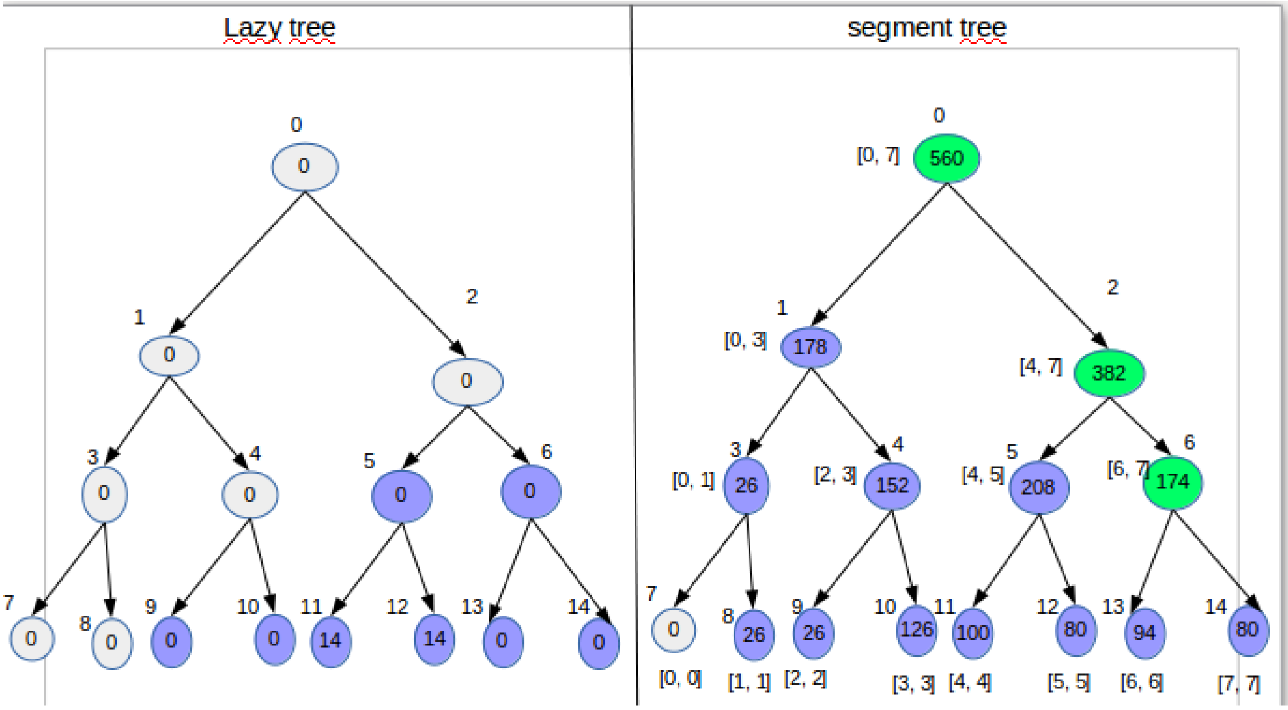
pos	lazy[pos]	Mis à jour ?	Overlap ?	Nœuds suivants
0	0	oui	partial	1 et 2
1	0	oui	no	2
2	0	oui	partial	5 et 6
5	0	oui	total	6
SegTree[5] += 2 * 14, lazy[11] = lazy[12] = 14				



pos	lazy[pos]	Mis à jour ?	Overlap ?	Nœuds suivants
6	0	oui	partial	13 et 14
13	0	oui	total	14
SegTree[13] += 1 * 14				



pos	lazy[pos]	Mis à jour ?	Overlap ?	Nœuds suivants
14	0	oui	no	aucun
Mettre à jour tout les parents				



<u>Query #6 : 1 4 8 => sum[3, 7] = ?</u>

pos	lazy[pos]	Mis à jour ?	Overlap ?	Nœuds suivants
0	0	oui	partial	1 et 2
1	0	oui	partial	3 et 4
3	0	oui	no	4
4	0	oui	partial	9 et 10
9	0	oui	no	10
10	0	oui	total	2
Return segTree[10] = 126				
2	0	oui	total	aucun
Return segTree[2] = 382				

Query #6 : 1 4 8 => sum[3, 7] = 126 + 382 = 508

Code complet C++

Submission link : http://www.spoj.com/submit/HORRIBLE/id=18947895
ideone link : http://ideone.com/FZ7IoD

```cpp
#include <cstdio>
#include <cstdlib>
#include <cmath>

typedef long long ll;

ll *segTree = NULL, *lazy = NULL;
ll T, N, C;

ll query(ll p, ll q, ll low, ll high, ll pos){
  if (low > high)
    return 0;
  //être sur que tout les propagations soient faites
  //à la position pos. (lazy[pos] == 0)
  //Sinon, mettre à jour le segment tree à pos et marquer
  //tout ses fils pour une lazy propagation.
  if(lazy[pos] != 0){
    segTree[pos] += (high - low + 1) * lazy[pos];

    //Not a leaf node
    if(low != high){
      lazy[pos * 2 + 1] += lazy[pos];
      lazy[pos * 2 + 2] += lazy[pos];
    }
    lazy[pos] = 0;
  }

  //No overlap
  if(q < low || p > high)
      return 0;

  //Total overlap
  if(p <= low && q >= high)
        return segTree[pos];

  //Partial overlap
  ll mid = (high + low) / 2;
  return query(p, q, low, mid, 2 * pos + 1) +
          query(p, q, mid + 1, high, 2 * pos + 2);
}
```

```
void update(ll p, ll q, ll v, ll low, ll high, ll pos){
  if (low > high)
    return;
  //être sur que tout les propagations soient faites
  //à la position pos. (lazy[pos] == 0)
  //Sinon, mettre à jour le segment tree à pos et marquer
  //tout ses fils pour une lazy propagation.
  if (lazy[pos] != 0) {
    segTree[pos] += (high - low + 1) * lazy[pos];

    if (low != high) {
      lazy[2 * pos + 1] += lazy[pos];
      lazy[2 * pos + 2] += lazy[pos];
    }

    lazy[pos] = 0;
  }

  //No overlap
  if ( q < low || p > high)
    return;

  //Total overlap
  if (p <= low && q >= high) {
    segTree[pos] += (high - low + 1) * v;

    //Not a leaf node
    if (low != high) {
      lazy[2 * pos + 1] += v;
      lazy[2 * pos + 2] += v;
    }

    return;
  }

  //Partial overlap
  int mid = (high + low) / 2;
  update(p, q, v, low, mid, 2 * pos + 1);
  update(p, q, v, mid + 1, high, 2 * pos + 2);

  segTree[pos] = segTree[2 * pos + 1] + segTree[2 * pos + 2];
}
```

```
int main() {
  scanf("%lld", &T);
  while(T--) {
    scanf("%lld%lld", &N, &C);

    input = new ll [N];


    ll SN = (ll)(ceil(log2(N)));
    SN = 2 * (1 << SN) - 1;
    segTree = new ll[SN];
    lazy = new ll[SN];

    for (ll i = 0 ; i < C ; ++i) {
      ll c;
      scanf("%lld", &c);
      if (c == 1) {
        ll p, q;
        scanf("%lld%lld",&p, &q);
        printf("%lld\n", query(p-1, q-1, 0, N-1, 0));
      }
      if (c == 0) {
        ll p, q, v;
        scanf("%lld%lld%lld",&p, &q, &v );
        update(p-1, q-1, v, 0, N-1, 0);
      }
      for (ll i = 0 ; i < SN ; ++i)
        printf("%lld ",segTree[i]);
      printf("\n");

      for (ll i = 0 ; i < SN ; ++i)
        printf("%lld ",lazy[i]);
      printf("\n");
    }
  }

  return 0;
}
```

18947895		2017-03-12 02:37:07	Horrible Queries	**accepted** edit ideone it	0.30	58M	C++ 4.3.2

Les Bitmasks

Motivation

Supposons que vous avez une liste d’objets et vous voulez sélectionner ou de ne pas sélectionner des objets parmi les objets de la liste. D’une autre manière, comment pourrons nous présenter un sous ensemble d’un ensemble.

Une manière de faire ça est d’utiliser un map (C++) ou un hashmap (Java) ou un dictionnaire (Python) est d’associer à chaque éléments un booléen pour indiquer si cette élément est choisi ou non. Cependant, ceci peut exploiter beaucoup de mémoire, surtout pour parcourir ces différents structures de données. **Si la taille de l’ensemble à parcourir n’est très grand**, un bitmask est la solution la plus efficace.

C’est quoi un bitmask ?

Tout entiers et caractères sont représentés sous forme d’une séquence binaire. Et cette dernière agit comment une séquence de valeurs booléennes lesquelles on peut les comparés avec une autre séquence binaire (Un bitmask, a.k.a. lightweight) via les bitwises (&, |, ^, <<, >>, ~).

Depuis cette séquence binaire, on peut savoir si un objet est choisi ou pas. Le premier bit (le LSB : least significant bit, celui le plus à droite.) donne l’information si le premier objet est choisi ou non, le second bit pour le second objet, etc.

Par exemple, si depuis un ensemble de 5 objets on a choisi le premier, le troisième et le quatrième, le bitmask pour représenter cette situation est 01101.

Manipulation des bitmask

Exemple #1 : savoir la casse d’une lettre

La représentation binaire des lettres majuscules :

lettre	En binaire
‘A’	0100 0001
‘B’	0100 0010
‘Z’	0101 1010

La représentation binaire des lettres minuscules:

lettre	En binaire
‘a’	0110 0001
‘b’	0110 0010
‘z’	0111 1010

On remarque que le bit n°6 est toujours 0 dans le case où la lettre est majuscule. Et 1 dans le case où elle est minuscule. Si on prend le mask 0010 0000 (32) et effectuer un "et binaire" (&) avec une lettre, on obtient :

- 0 si la lettre est majuscule.
- ≠ 0 si la lettre est minuscule.

	‘Z’	01011010		‘z’	01111010
&	bitmask	00100000	&	bitmask	00100000
	0	0000000		Non-0	00100000

Code C++ :

```
#include <bits/stdc++.h>

using namespace std;

int main() {
  char c;
  cin >> c;

  //if ( (c & 32) == 0)
  //if ( (c & 0x20) == 0)
  if ( (c & 0b00100000) == 0)
    cout << "Uppercase.\n";
  else
    cout << "Lowercase.\n";
  return 0;
}
```

Exemple #2 : Conversion de la casse d’une lettre

On utilise *XOR* et le bitmask 00100000 sur la lettre à modifier :

	‘Z’	01011010		‘z’	01111010
^	bitmask	00100000	^	bitmask	00100000
	‘z’	01111010		‘Z’	01011010

Exemple #3 : Multiplier un entier par 2

On applique un décalage à logique qui consiste à supprimer un bit d'un côté du vecteur pour le remplacer par un zéro de l'autre côté.
On effectue un décalage à gauche de 1 bit.
13 = 00001101
13 << 1 = 00001101 << 1 = 00011010 = 26.

Exemple #4 : Diviser un entier par 2

On effectue un décalage à droite de 1 bit.
17 / 2 = 17 >> 1 = 00010001 >> 1 = 00001000 = 8

Exemple #5 : Calculer le nombre des sous ensembles qu'on peut former à partir d'un ensemble de cardinalité *n*

Le nombre des sous ensembles qu'on peut former d'un ensemble de cardinalité n est 2^n, qu'on peut calculer par **1<< n**.

Exemple #6 : Ajouter le *j^éme^* objet au sous ensemble *A*

Mettre le $j^{éme}$ bit à 1.
<u>Principe :</u>
1- Décaler l'entier 1 à gauche de j bits.
2- Faire un OU binaire entre l'ensemble est résultat obtenu en 1.
Formule : A |= (1 << j)

<u>Exemple</u> : si on veut ajouter l'objet 6 au sous ensemble qui contient {1,4}
A = 00001001
1 << 6 = 00000001 << 6 = 01000000
A | 00100000 = 00001001 | 01000000 = 0**1**001001
Le sous ensemble *A* contient, Maintenant, {1,4,7}

Exemple 7 : Supprimer le $j^{éme}$ objet du *sous ensemble A*

Mettre le $j^{éme}$ bit à 0
<u>Principe :</u>
1- Décaler l'entier 1 à gauche de j bits.
2- Appliquer un non binaire au résultat obtenu en 1.
3- Faire un et binaire entre l'ensemble est le résultat obtenu en 2.
Formule : A &= ~(1 << j)

<u>Exemple</u> : si on veut supprimer l'objet 6 du sous ensemble qui contient {1,4,7}
A = 01001001
1 << 6 = 00000001 << 6 = 01000000
~01000000 = 10111111
A & 00100000 = 01001001 | 10111111 = 0**0**001001
Le sous ensemble A contient, Maintenant, {1,3}

Exemple 8 : Vérifier si le *j^éme^* objet appartient *au sous ensemble A*

Vérifier si le $j^{éme}$ bit est 1.

Principe :

1- Décaler l'entier 1 à gauche de j bits.

2- Faire un et binaire entre l'ensemble est le résultat obtenu en 1.

Formule : verif = A & (1 << j)

Si verif == 0 alors que le $j^{éme}$ objet n'existe pas.

Si verif != 0 alors le $j^{éme}$ objet existe.

Exemple :

Soit une liste de N entiers, on veut faire la somme de tous les sous ensembles formés à partir des entier de cette liste.

Pour L = {2,6,11}

Le programme affiche :

{2}: 2

{6}: 6

{2, 6}: 8

{11}: 11

{2, 11}: 13

{6, 11}: 17

{2, 6, 11}: 19

On peut former 2^3 sous ensembles.

Pour ce faire, on a parcourir tout les sous ensembles, qu'on va nommé *i*, et pour chaque éléments à la position *j* de l'ensemble, vérifier si *j* existe dans le sous ensemble *i*. (voir tableau ci-dessous)

i	j	i & (1 << j)		somme	résultat
0	0	0 & 1 = 0	Pas de sous ensemble.	0	
	1	00 & 10 = 0		0	
	2	000 & 100 = 0		0	
1	0	1 & 1 != 0	**{2}**	2	
	1	01 & 10 = 0		0	
	2	001 & 100 = 0		0	**{2} : 2**
2	0	10 & 01 = 0		0	
	1	10 & 10 != 0	**{6}**	6	
	2	010 & 100 = 0		0	**{6} : 6**
3	0	11 & 01 != 0	{2}	2	
	1	11 & 10 != 0	**{2,6}**	8	
	2	011 & 100 = 0		8	**{2,6} : 8**
4	0	100 & 001 = 0		0	
	1	100 & 010 = 0		0	
	2	100 & 100 != 0	**{11}**	11	**{11} : 11**
5	0	101 & 001 != 0	{2}	2	
	1	101 & 010 = 0		2	
	2	101 & 100 != 0	**{2,11}**	13	**{2,11} : 13**
6	0	110 & 001 = 0		0	
	1	110 & 010 != 0	{6}	6	
	2	110 & 100 != 0	**{6,11}**	17	**{6,11} : 17**
7	0	111 & 001 != 0	{2}	2	
	1	111 & 010 != 0	{2,6]	8	
	2	111 & 100 != 0	**{2,6,11}**	19	**{2,6,11} : 19**

Code C++ : $O(2^n * n)$

```
#include <bits/stdc++.h>
using namespace std;

void display(vector <int > v, int n) {
  for (auto vi: v)
    cout << vi << ' ';
  cout << '\n';
}

void fillArr (vector <int> & v, int n) {
  for (int i = 0 ; i < n ; ++i) {
    int vi;
    cin >> vi;
    v.push_back(vi);
  }
}

void allSubsets(vector <int> v, int n) {
  int numberSubsets = 1 << n;
  for (int i = 0 ; i < numberSubsets ; ++i) {
    int s = 0;
    cout << '{';
    for (int j = 0 ; j < n ; ++j) {
      if ( (i & (1 << j)) != 0 ) {
        s += v[j];
        cout << v[j] <<  ", ";
      }
    }
    cout <<'}';
    cout << ": " << s;
    cout << '\n';
  }
}
int main() {
  int n;
  cin >> n;

  vector <int> v;
  fillArr(v, n);
  cout << '\n';
  display(v, n);
  cout << '\n';
  allSubsets(v, n);

  return 0;
}
```

Problème : Sujet "Pizza" de la 2e tour de la TOP 2017 (top17c2p5.pdf)

On prend l'exemple #3 du sujet :

Exemple Input 3

```
3 3
1 2
1 3
2 3
```

Exemple Output 3

```
4
```

Explication exemple 3

Il est possible soit de faire une pizza sans ingrédients ou bien avec un seul.

Soit S l'ensemble des ingrédients qu'on a : S = {1, 2, 3}

On peut faire une pizza :

- Sans ingrédients : {}
- Avec l'ingrédient :
  - ✗ {1} tous seul.
  - ✗ {2} tous seul
  - ✗ {3} tous seul

Mais pas avec les ingrédients :

- {1, 2}
- {2, 3}
- {1, 2, 3}

Pour calculer le nombre des pizzas qu'on puisse réaliser en mélangeant des différentes combinaisons des ingrédients, et en respectant les interdictions du mélange des ingrédients, on vérifie si les ingrédients *a* et *b* à, ne pas mélanger, existent dans les différents ensembles des ingrédients. S'il existent alors on ajouter rien au nombre, sinon le nombre des pizzas est incrémenté de 1.

Exemple :

S = {1, 2, 3}

À ne pas mélanger : {1, 2}, {1, 3} et {2, 3}

ensembles	Nombre des pizzas qu'on puisse faire
	0
{}	1
{1}	2
{2}	3
{1, 2}	3
{3}	4
{1, 3}	4
{2, 3}	4
{1, 2, 3}	4

Algorithmiquement parlant, on utilise les bitmaks pour pouvoir résoudre ce problème. La complexité temporelle est $O(2^N * M)$, ce qui est énorme si *N* est très grand.

On stocke les ingrédients interdits dans deux tableaux *a* et *b* :

Les ingrédients à ne pas mélanger sont :

{1, 2} => a[0] = 0 et b[0] = 1

{1, 3} => a[1] = 0 et b[1] = 2

{2, 3} => a[2] = 1 et b[2] = 2

a		
0	1	2
0	0	1

b		
1	2	2
1	2	2

Puis, pour chaque ensemble *i* générer on vérifie si le *a[j]*^{eme} et *b[j]*^{eme} bit sont à HIGH (égale à 1) dans *i*.

i	i en binaire	ensemble	j	a[j] et b[j] existent-t-ils dans l'ensemble ?	Nombre des pizzas qu'on puisse faire
					0
0	0000	{}	0	non	0
			1	non	0
			2	non	0
					1
1	0001	{1}	0	non	1
			1	non	1
			2	non	1
					2
2	0010	{2}	0	non	2
			1	non	2
			2	non	2
					3
3	0011	{1, 2}	0	oui	3, On ajoute 0 au nombre et on quitte la boucle.
			1		
4	0100	{3}	0	non	
			1	non	
			2	non	
					4
5	0101	{1, 3}	0	non	
			1	oui	4, On ajoute 0 au nombrer et on quitte la boucle.
					4
6	0110	{2, 3}	0	non	
			1	non	
			2	oui	4, On ajoute 0 au nombrer et on quitte la boucle.
					4
7	0111	{1, 2, 3}	0	oui	4, On ajoute 0 au nombrer et on quitte la boucle.
					4

Code C++ : Solution proposée par M. Firas Trabelsi

```
#include <cstdio>

const int MAX_N = 20;

int a[MAX_N], b[MAX_N];

int countSubsets(int n, int m) {
  int numberSubsets = 1 << n;
  int ans = 0;
  for (int i = 0 ; i < numberSubsets ; ++i) {
    int s = 1;
    for (int j = 0 ; j < m ; ++j) {
      if ( (i & (1 << a[j])) && (i & (1 << b[j])) ) {
        s = 0;
        break;
      }
    }
    ans += s;
  }
  return ans;
}

int main() {

  int N, M;
  scanf("%d%d", &N, &M);
  if (M == 0)
    printf("%d\n", 1 << N);
  else {
    for (int i = 0 ; i < M ; ++i) {
      scanf("%d%d", &a[i], &b[i]);
      a[i]--;
      b[i]--;
    }
    printf("%d\n", countSubsets(N,M) );
  }
  return 0;
}
```

Problème : candles-2

https://www.hackerrank.com/challenges/candles-2

Pour résoudre de problème, il faut les acquis suivants : dp, bitmask, inclusion-exclusion et BIT..
(Greetz to ***cherim_*** from ##algorithms on IRC server : freenode.net)

La complexité temporelle ciblée = $O(2^k N \log N)$

Étape #1 : calcul du nombre des sous-séquences croissante en hauteur indépendamment des couleurs

Un *mask:* est la séquence en hauteur construit à partir uniquement d'une (des) certaine(s) couleur(s).

f(mask) : est nombre des séquences croissantes par hauteur construisent à partir de *mask*.

Tout d'abord, Pour chaque *mask*, nous calculons le nombre de sous-séquences croissantes par hauteur contenant uniquement les couleurs présentent dans le *mask*. (notée *f(mask)*)
Pour l'exemple :
4 3
1 1
3 2
2 2
4 3

Si le *mask* = 6 (0110), *f(mask) = 5*. Parce que les couleurs impliquées dans *mask* sont {2, 3} donc la séquence des hauteurs qu'en résulte est {3, 2, 4}. Les séquences croissantes par hauteur sont : {3}, {2}, {4}, {3, 4} et {2, 4}.

On sait que : $\forall mask, on\, a, f(mask) \geq 1$

$$f(mask) = 1 + \sum_{i=0}^{n-1} f(i), h_i < h_n$$

$avec\, f(i)\, est$ le $nombre\, des\, séquences\, croissantes\, en\, hauteur\, se\, terminant\, à\, la\, couleur\, d'indice\, i$

Exemple :

H			
0	**1**	**2**	**3**
1	3	2	4

C			
0	**1**	**2**	**3**
1	2	2	3

mask	mask en binaire	Couleurs présentent dans le mask	La séquence des hauteurs qu'en résulte	Les séquences croissantes en hauteurs i = indice : les séquences croissante par hauteur se terminant à la couleur d'indice i => f(i) = # les séquences croissante par hauteur se terminant par la couleur d'indice i	f(mask)
0	0000	--	--	--	0
1	0001	{1}	{1}	i = 0 : {1} => **f(0) = 1**	1
2	0010	{2}	{3, 2}	i= 1 : {3} ⇒ **f(1) = 1** i = 2 : {2} => **f(2) = 1**	2
3	0011	{1, 2}	{1, 3, 2}	i = 0 : {1} => **f(0) = 1** i = 1 : {3}, {1, 3} => **f(1) = 2** i = 2 : {2}, {1, 2} => **f(2) = 2**	5
4	0100	{3}	{4}	i = 3 : {4} => **f(3) = 1**	1
5	0101	{1,3}	{1, 4}	i = 0 : {1} => **f(0) = 1** i = 3 : {4}, {1, 4} => **f(3) = 2**	3
6	0110	{2, 3}	{3, 2, 4}	i = 1 : {3} => **f(1) = 1** i = 2 : {2} => **f(2) = 1** i = 3 : {4}, {3, 4}, {2, 4} => **f(3) = 3**	5
7	0111	{1,2,3}	{1,3,2,4}	I = 0 : {1} => **f(0) = 1** i = 1 : {3}, {1, 3} => **f(1) = 2** i = 2 : {2}, {1, 2} = >**f(2) = 2** i = 3 : {4}, {1, 4}, {3, 4}, {2, 4}, {1, 3, 4}, {1, 2, 4} => **f(3) = 6**	11

Le mask #7, nous donne la somme de toutes les séquences croissantes en hauteurs indépendamment des couleurs.

Nous pouvons calculer directement (par brute force) la $\sum_{i=0}^{n-1} f(i)$ pour chaque *mask*, mais ça coûtera O(N²) (La complexité total sera O(2^k * N²)). Nous pouvons utiliser une structure de donnée qui garde les sommes calculées : **Une fenwick tree ou une segment tree**. En utilisant une telle structure, on peut rapidement calculer la somme de toutes les *f(i)* (les séquences croissantes par hauteur se terminant à la couleur d'indice *i*)

$$f(i)=1+query(h_i-1),\ \ 0\leq i<n$$

$$et\,(f(i)\,est\ \mathrm{le}\ nombre\,des\,séquences\,croissante\,par\,hauteur\,se\,terminant\,à\,la\,couleur\,d'indice\,i)$$

$$f(mask)=\sum_{i=0}^{n-1} fi=\sum_{i=0}^{n-1}(1+query(h_i-1))$$

$$Après\,chaque\,calcul\,de\,f(i),on\,met\,à\ jour\leq Fenwick\,tree\,à\,la\,position\,H_i:update(h_i)$$

Exemple : Pour savoir les nœuds suivant et précédents d'un nœud de BIT, consulter l'annexe : **BIT-NEXTS-PREVS**

H			
0	**1**	**2**	**3**
1	3	2	4

C			
0	**1**	**2**	**3**
1	2	2	3

ft					**mask**	**f(i)**	**f(mask)**
0	**1**	**2**	**3**	**4**			
					1 (0001)		
0	0	0	0	0		i = 0 => f(0) = 1 + query(H[0] - 1) = 1 + query(0) = 1 + 0 = 1	
0	1	1	0	1		update(H[0]) = update(1)	
							1
					2 (0010)		
0	0	0	0	0		i = 1 => f(1) = 1 + query(H[1] - 1) = 1 + query(2) = 1 + 0 = 1	
0	0	0	1	1		update(H[1]) = update(3)	
							1
0	0	0	1	1		i = 2 => f(2) = 1 + query(H[2] - 1) = 1 + query(1) = 1 + 0 = 1	
0	0	1	1	2		update(H[2]) = update(2)	
							2

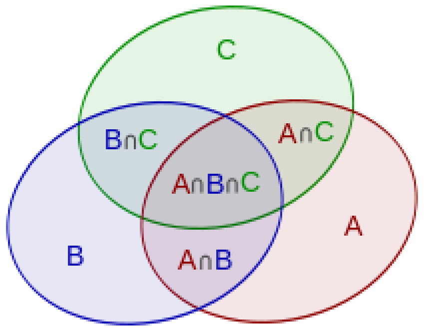
					3(0011)		
0	0	0	0	0		i = 0 => f(0) = 1 + query(H[0] - 1) = 1 + query(0) = 1 + 0 = 1	
0	1	1	0	1		update(H[0]) = update(1)	
							1
0	1	1	0	1		i = 1 => f(1) = 1 + query(H[1] - 1) = 1 + query(2) = 1 + 1 = 2	
0	1	1	1	3		update(H[1]) = update(3)	
							3
0	1	1	1	3		i = 2 => f(2) = 1 + query(H[2] - 1) = 1 + query(1) = 1 + 1 = 2	
0	1	3	1	5		update(H[2]) = update(2)	
							5
					4(0100)		
0	0	0	0	0		i = 3 => f(3) = 1 + query(H[3] - 1) = 1 + query(3) = 1 + 0 = 1	
0	0	0	0	1		update(H[3]) = update(4)	
							1
					5(0101)		
0	0	0	0	0		i = 0 => f(0) = 1 + query(H[0] - 1) = 1 + query(0) = 1 + 0 = 1	
0	1	1	0	1		update(H[0]) = update(1)	
							1
0	1	1	0	1		i = 3 => f(3) = 1 + query(H[3] - 1) = 1 + query(3) = 1 + 1 = 2	
0	1	1	2	3		update(H[3]) = update(4)	
							3
					6(0110)		
0	0	0	0	0		i = 1 => f(1) = 1 + query(H[1] - 1) = 1 + query(2) = 1 + 0 = 1	
0	0	0	1	1		update(H[1]) = update(3)	
							1
0	0	0	1	1		i = 2 => f(2) = 1 + query(H[2] - 1) = 1 + query(1) = 1 + 0 = 1	
0	0	1	1	2		update(H[2]) = update(2)	
							2
0	0	1	1	2		i = 3 => f(3) = 1 + query(H[3] - 1) = 1 + query(3) = 1 + 2 = 3	
0	0	1	1	5		update(H[3]) = update(4)	
							5
					7(0111)		
0	0	0	0	0		i = 0 => f(0) = 1 + query(H[0] - 1) = 1 + query(0) = 1 + 0 = 1	
0	1	1	0	1		update(H[0]) = update(1)	

							1
0	1	1	0	1		i = 1 => f(1) = 1 + query(H[1] - 1) = 1 + query(2) = 1 + 1= 2	
0	1	1	2	3		update(H[1]) = update(3)	
							3
0	1	1	2	3		i = 2 => f(2) = 1 + query(H[2] - 1) = 1 + query(1) = 1 + 1 = 2	
0	1	3	2	5		update(H[2]) = update(2)	
							5
0	1	3	2	5		i = 3 => f(3) = 1 + query(H[3] - 1) = 1 + query(3) = 1 + 5 = 6	
0	1	3	2	11		update(H[3]) = update(4)	
							11

Étape #2 : calcul du nombre des sous-séquences croissantes contenant toutes les couleurs

Une fois le nombre des sous-séquences croissantes en hauteurs est calculé, nous appliquons le **principe d'inclusion-exclusion.** (https://en.wikipedia.org/wiki/Inclusion%E2%80%93exclusion_principle)

Exemple : pour K = 3



Pour trouver **le nombre de séquences croissantes en hauteurs qui contenant toutes les couleurs** (|**A**∩**B**∩**C**|) :
On a le nombre de toutes les séquences indépendamment des couleurs (qui représente tout la surface de la figure) : 11. À partir de lequel, **on réduit les nombres des séquences avec une seule couleur non contenue (-|A∩B|-|A∩C|-|B∩C|)**. Et **on y ajoute les nombres des séquences les nombres des séquences avec deux couleurs non contenus (+|A|+|B|+|C|)**

Soit :
S, l'ensemble de toutes les séquences croissantes en hauteur indépendamment des couleurs :
S = {{1}, {3}, {2}, {4}, {1, 3}, {1, 2}, {1, 4}, {3, 4}, {2, 4}, {1, 3, 4}, {1, 2, 4}}

Soit A, l'ensemble des séquences dont uniquement les couleurs 1 et 2 non contenues
A = {{4}}

Soit B, l'ensemble des séquences dont uniquement les couleurs 1 et 3 non contenues
B = {{3}, {2}}

Soit C, l'ensemble des séquences dont uniquement les couleurs 2 et 3 non contenues
C = {{1}}

A∩B, l'ensemble des séquences dont uniquement la couleur 1 non contenue
A∩B = {{3}, {2}, {4}, {3, 4}, {2, 4}}

A∩C, l'ensemble des séquences dont uniquement la couleur 2 non contenue
A∩C = {{1}, {4}, {1, 4}}

B∩C, l'ensemble des séquences dont uniquement la couleur 3 non contenue
B∩C = {{1}, {3}, {2}, {1, 2}, {1, 3}}

$|A \cap B \cap C| = |S| + |A| + |B| + |C| - |A \cap B| - |A \cap C| - |B \cap C| = 11 + 1 + 2 + 1 - 5 - 3 - 5 = 2$
Nous avons deux séquences croissantes en hauteurs contenant toutes les couleurs.

Comment effectuer l'inclusion-l'exclusion

Le *mask*, déjà calculé à l'étape #1, nous indique quelle est l'ensemble à inclure ou à exclure, en regardant le nombre des bits élevés (set bits).

Revenons à notre exemple :

mask	mask en binaire	Les séquences croissantes en hauteurs	f(mask)	À inclure-exclure	Progression du calcul du résultat f(n)
0	0000	--	0		0
1	0001	{1}	1	+|C| = 1	1
2	0010	{3}, {2}	2	+|B| = 2	3
3	0011	{1}, {3}, {2}, {1, 2}, {1, 3}	5	-|B∩C| = -5	-2
4	0100	{4}	1	+|A| = 1	-1
5	0101	{1}, {4}, {1, 4}	3	-|A∩C| = -3	-4
6	0110	{3}, {2}, {4}, {3, 4}, {2, 4}	5	-|A∩B| = -5	-9
7	0111	{1}, {3}, {2}, {4}, {1, 3}, {1, 2}, {1, 4}, {3, 4}, {2, 4}, {1, 3, 4}, {1, 2, 4}	11	+|S| = +11	**<u>2</u>**

À la fin le résultat final est cumulé de la manière suivante :
pour chaque *mask*, si nombre des bits élevés du *mask* modulo 2 est égale *K* modulo 2 alors on ajoute au résultat la valeur de *f(n)*. Sinon on en réduit.

Calcul de *fn* final : Pseudo-code

```
si setBits(mask) modulo 2 = K modulo 2 :
        fn ← fn + fmask
sinon :
        fn ← fn - fmask
```

Autre exemple : N = 6 et K = 4

6 4
1 2
7 1
2 2
9 3
10 4
8 3

H					
0	**1**	**2**	**3**	**4**	**5**
1	7	2	9	10	8

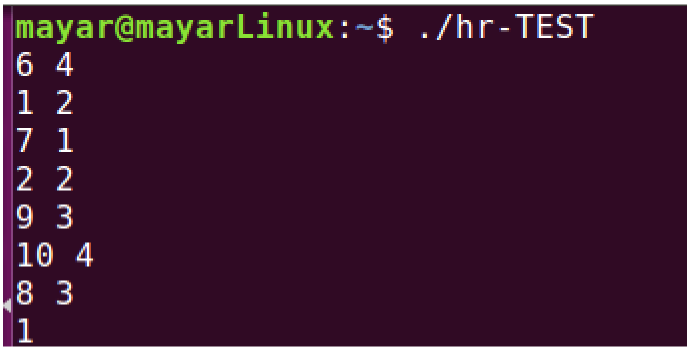
C					
0	**1**	**2**	**3**	**4**	**5**
2	1	2	3	4	3

mask	mask en binaire	Couleurs présentent dans le mask	La séquence des hauteurs qu'en résulte	f(i)	f(mask)	setBits(f(mask)) modulo 2 = K modulo 2	f(n)
0	0000	--	--	--	0	oui	0
1	0001	{1}	{7}	i = 0 : {7} => **f(0) = 1**	1	non	-1
2	0010	{2}	{1, 2}	i= 0 : {1} ⇒ **f(0) = 1** i = 2 : {2}, {1, 2} => **f(2) = 2**	3	non	-4
3	0011	{1, 2}	{1, 7, 2}	i = 0 : {1} => **f(0) = 1** i = 1 : {1}, {1, 7} => **f(1) = 2** i = 2 : {2}, {1, 2} => **f(2) = 2**	5	oui	1
4	0100	{3}	{9, 8}	i = 3 : {9} => **f(3) = 1** i = 5 : {8} => **f(5) = 1**	2	non	-1
5	0101	{1,3}	{7, 9, 8}	i = 1 : {7} => **f(1) = 1** i = 3 : {9}, {7, 9} => **f(3) = 2** i = 5 : {8}, {7, 8}	5	oui	4

				=> **f(5) = 2**			
6	0110	{2, 3}	{1, 2, 9, 8}	i = 0 : {1} => **f(0) = 1** i = 2 : {2}, {1, 2} => **f(2) = 2** i = 3 : {9}, {1, 9}, {2, 9}, {1, 2, 9} => **f(3) = 4** i = 5:{8}, {1, 8}, {2, 8}, {1, 2, 8} => **f(5) = 4**	11	oui	15
7	0111	{1, 2, 3}	{1,7, 2, 9, 8}	i = 0:{1} => **f(0) = 1** i = 1:{7}, {1, 7} => **f(1) = 2** i = 2 : {2}, {1, 7}, {1, 2} => **f(2) = 3** i = 3 : {9}, {1, 9}, {7, 9}, {2,9}, {1, 2, 9}, {1, 7, 9} => **f(3) = 6** i = 5 : {8}, {1, 8}, {7, 8}, {2, 8}, {1, 2, 8}, {1, 7, 8} => **f(5) = 6**	18	non	-3
8	1000	{4}	{10}	i = 4 : {10} => f(4) = 1	1	non	-4
9	1001	{1, 4}	{7, 10}	i = 1:{7} => f(1) = 1 i = 4 : {10}, {7, 10} => f(4) = 2	3	oui	-1
10	1010	{2, 4}	{1, 2, 10}	i = 0:{1} => f(0) = 1 i = 2:{2}, {1, 2} => f(2) = 2 i = 4:{10}, {1, 10}, {2, 10}, {1, 2, 10} => f(4) = 4	7	oui	6
11	1011	{1, 2, 4}	{1, 7, 2, 10}	i = 0:{1} => f(0) = 1 i = 1:{7}, {1, 7} => f(1) = 2 i = 2:{2}, {1, 2} => f(2) = 2 i = 4 : {10}, {1,	11	non	-5

				10}, {7, 10}, {2, 10}, {1, 7, 10}, {1, 2, 10} => f(4) = 6			
12	1100	{3, 4}	{9, 10, 8}	i = 3:{9} => f(1) = 1 i = 4:{10}, {9, 10} => f(4) = 2 i = 5:{8} => f(5) = 1	4	oui	**-1**
13	1101	{1, 3, 4}	{7, 9, 10, 8}	i = 1:{7} => f(1) = 1 i = 3:{9}, {7, 9} => f(3) = 2 i = 4:{10}, {7, 10}, {9, 10}, {7, 9, 10} => f(4) = 4 i = 5 : {8}, {7, 8} => f(5) = 2	9	non	-10
14	1110	{2, 3, 4}	{1, 2, 9, 10, 8}	i = 0 : {1} => f(1= = 1 i= 2:{2}, {1, 2} => f(2) = 2 i = 3:{9}, {1, 9}, {2, 9}, {1, 2, 9} => f(3) = 3 i = 4:{10}, {1, 10}, {2, 10}, {9, 10}, {1, 2,10}, {1, 9, 10}, {1, 2, 9, 10} => f(4) = 7 i = 5:{8}, {1, 8}, {2, 8}, {1, 2, 8} => f(5) = 4	17	non	-27
15	1111	{1,2,3,4}	{1, 7, 2, 9, 10, 8}	i =0:{1} => f(1) = 1 i = 1 : {7}, {1, 7} => f(1) = 2 i = 2:{2}, {1, 2} => f(2) = 2 i = 3 : {9}, {1, 9}, {7, 9], {2, 9}, {1, 7, 9}, {1, 2, 9} => f(3) = 6 i = 4 : {10}, {1, 10}, {7, 10}, {2, 10}, {9, 10}, {1, 7,	28	oui	**1**

				10}, {1, 2, 10}, {1, 9, 10}, {1, 7, 9, 10}, {2, 9, 10}, {7, 9, 10} => f(4) = 11, i = 5:{8}, {1, 8}, {7, 8}, {2, 8}, {1, 7, 8}, {1, 2, 8} => f(5) = 6			

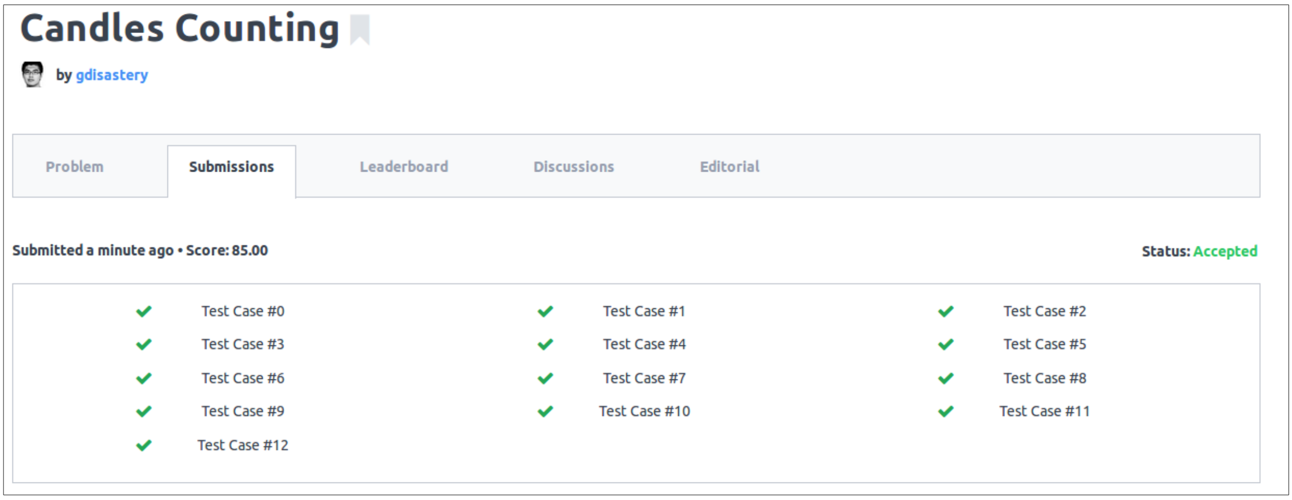


Une dernière chose, pour savoir les couleurs qui interviennent dans un *mask* donné, on utilise la formule ***mask >> (C[i] - 1)) & 1***. Si elle donne 1, c'est que le *mask* contienne un couleur à la position *i*.

Problem : https://www.hackerrank.com/challenges/candles-2
Submission : https://www.hackerrank.com/challenges/candles-2/submissions/code/36868337
C++ code :

```
/*
Task: https://www.hackerrank.com/challenges/candles-2
Lang: C++
Required Knowledge: dp, bitmask, inclusion-exclusion, BIT
Time complexity: O (2^k N log N)
*/

#include <cstdio>
#include <cstring>
const int MAX_N = 50000;
const int MAX_K = 7;
const int MAX_H = 50000;

// modulo code
const int mod = 1e9 + 7;
void madd(int& a, int b){ a += b;  if (a >= mod) a -= mod; }

// Fenwick code //
int ft[MAX_N + 1];
```

```
void update(int i, int x) {
  for (; i <= MAX_N; i += (i & -i))
    madd(ft[i], x);
}
int query(int i) {
  int s = 0;
  for (; i > 0; i -= (i & -i))
    madd(s, ft[i]);
  return s;
}

int main(){
  int N, K;
  int H[MAX_N], C[MAX_N];

  scanf("%d%d", &N, &K);
  for (int i = 0; i < N; i++)
    scanf("%d%d", H + i, C + i);

  int fn = 0;
  for (int mask = 0; mask < (1 << K); mask++){
    memset(ft, 0, sizeof(ft));

    // Count number of sub-sequences of given ‘mask‘
    int fmask = 0;
    for (int i = 0; i < N; i++){
      if ((mask >> (C[i] - 1)) & 1){
        int fi = 1 + query(H[i] - 1);
        update(H[i], fi);
        madd(fmask, fi);
      }
    }

    // Inclusion-Exclusion Principle
    if (__builtin_popcount(mask) % 2 == K % 2)
      madd(fn, fmask);
    else
      madd(fn, mod - fmask);
  }

  printf("%d\n", fn);
  return 0;
}
```


Candles Counting
by gdisastery
Problem
Submissions
Leaderboard
Discussions
Editorial
Submitted a minute ago • Score: 85.00
Status: Accepted
Test Case #0
Test Case #1
Test Case #2
Test Case #3
Test Case #4
Test Case #5
Test Case #6
Test Case #7
Test Case #8
Test Case #9
Test Case #10
Test Case #11
Test Case #12

Annexes

MSE06H-Editorial

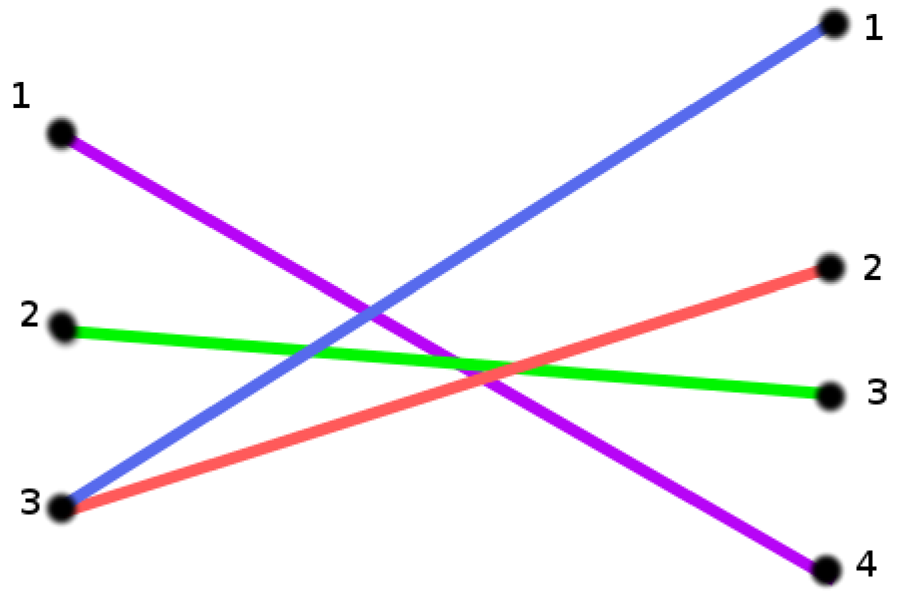
Solution of Satya-Jain : https://www.quora.com/How-do-I-solve-SPOJ-com-Problem-MSE06H-using-BIT/answer/Satya-Jain

> Japan plans to welcome the ACM ICPC World Finals and a lot of roads must be built for the venue. Japan is tall island with N cities on the East coast and M cities on the West coast (M ≤ 1000, N ≤ 1000). K superhighways will be build. Cities on each coast are numbered 1, 2, ... from North to South. Each superhighway is straight line and connects city on the East coast with city of the West coast.
>
> The funding for the construction is guaranteed by ACM. A major portion of the sum is determined by the number of crossings between superhighways. At most two superhighways cross at one location. Write a program that calculates the number of the crossings between superhighways.
>
> ## Input
>
> The input file starts with T - the number of test cases. Each test case starts with three numbers – N, M, K. Each of the next K lines contains two numbers – the numbers of cities connected by the superhighway. The first one is the number of the city on the East coast and second one is the number of the city of the West coast.
>
> ## Output
>
> For each test case write one line on the standard output:
>
> Test case "case number": "number of crossings"
>
> ## Sample

```
Input:
1
3 4 4
1 4
2 3
3 2
3 1

Output:
Test case 1: 5
```

Notre but est de calculer le nombre d'intersection entre les routes.
Dans l'exemple du problème, on a, ***N = 3***, **M = 4**, ***K = 4*** et l'ensemble des routes {{1, 4}, {2, 3}, {3, 2}, {3, 1}}

L’illustration de l’input donne :



Le nombre d’intersection est égale à 5.

L’approche naïve : Brute force

Stocker les couples de routes :

roads			
x = 1 y = 4	x = 2 y = 3	x = 3 y = 2	x = 3 y = 1
0	1	2	3

Pour chaque route ***i***, voir s’il intersecte avec une route ***j***, $0 \le i < k$ et $i+1 \le j < k$.
Une route(x, y) intersecte une route(a, b), sii (x < a et y > b) ou (x > a et y < b).

Pseudo-code :

```
ans ← 0
for i ∈ [0, k-1[ :
        for j ∈ [i+1, k[ :
                if (roads[i].x< roads[j].x et roads[i].y > roads[i].y ou roads[i].x > roads[i].x et 
roads[i].y < roads[i]).y :
                        ans ← ans + 1
```

La complexité temporelle est $O(k^2)$

L’approche intelligente : utiliser un arbre de Fenwick

Tout d’abord, il faut trier dans l’ordre croissant selon le premier indice. Si les deux premiers indices sont égaux, alors on trie les routes selon le deuxième indice dans l’ordre croissant.

Le tableau non trié :

roads			
x = 1 y = 4	x = 2 y = 3	x = 3 y = 2	x = 3 y = 1
0	1	2	3

Le tableau trié :

roads			
x = 1 y = 4	x = 2 y = 3	x = 3 y = 1	x = 3 y = 2
0	1	2	3

Pour trouver le nombre d’intersection, il suffit de calculer, pour chaque route ***i* ∈ [0, *k* [**, combien de route ***j* ∈[0, *i* [** ont $y_j > y_i$.

Il peut vous venir à l’esprit ce pseudo-code :

```
ans ← 0
for i ∈ [0, k[ :
        for j ∈ [0, i[ :
                if roads[j].y > roads[i].y :
                        ans ← ans + 1
```

La complexité temporelle est $O(k^2)$

Exécution à la main :

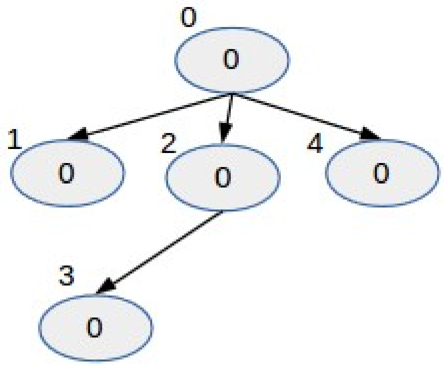
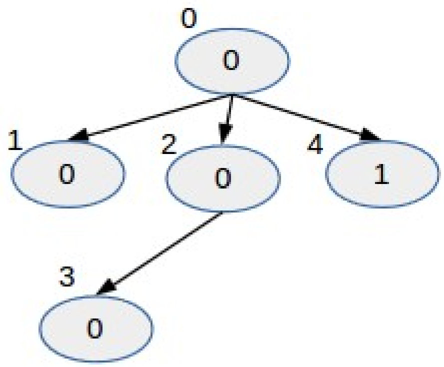
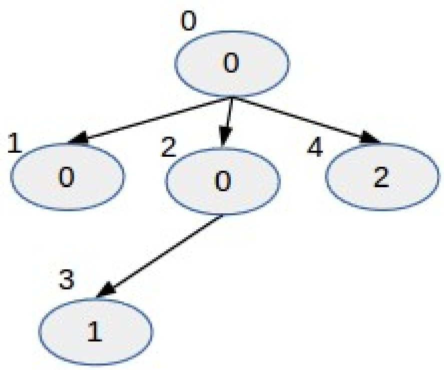
ans	i	j	roads[j].y	roads[i].y
0				
	0			
	1			
1		0	4	3
	2			
2		0	4	1
3		1	3	1
	3			
4		0	4	2
5		1	3	2
		2	1	2
On a 5 intersections.				

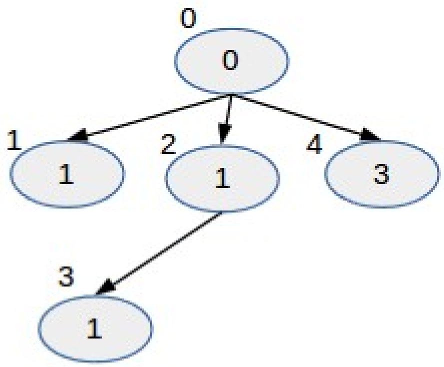
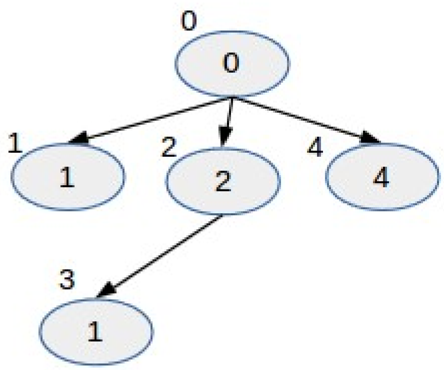
- L’arbre de Fenwick va maintenir le calcul et la mise à jour du nombre intersections.
- Les indices des nœuds de l’arbre sont les deuxièmes indices (les y_s) des routes.
- Le nombre d’intersections est le cumul des sommes de la range query :

  query(m)-query(roads[i].y)
- Après chaque range query on met à jour le nombre d’intersection pour chaque roads[i].y :

  update(roads[i].y, m).

avec : 0 ≤ i < k et m est le nombres des routes et m est le nombre des cités de la west coast.

	y	**0**	**1**	**2**	**3**	**4**	**BIT**	**ans**
		0	0	0	0	0		0
i = 0		0	0	0	0	1	query(4) – query(4) = 0 – 0 = 0 update(4) 	0
i = 1		0	0	0	1	2	query(4) – query(3) = 1 – 0 = 1 update(3) 	1
		0	1	1	1	3	query(4) – query(1) = 2 – 0 = 2 update(1)	3

i = 2								
i = 3		0	1	2	1	4	query(4) – query(2) = 3– 1 = 2 update(1) 	5

Problem : http://www.spoj.com/problems/MSE06H/
Submission : http://ideone.com/S20k7Y
C++ code :

```
#include <bits/stdc++.h>
using namespace std;

struct road {
    int x, y;
};

int tree[1009];

long long query(int pos){
    long long count = 0;
    while(pos) {
        count += tree[pos];
        pos -= (pos & -pos);
    }
    return count;
}

void update(int pos ,int MAX){
    while(pos <= MAX){
        tree[pos]+=1;
        pos += (pos & -pos);
    }
}

bool compare(const road &r1 ,const road &r2){
    return r1.x == r2.x ? r1.y < r2.y : r1.x < r2.x;
}
int main(){
    int t;
    scanf("%d",&t);
    for(int l  =1 ; l<=t ; l++){
        int n , m ,k ,x, y ;
        scanf("%d%d%d",&n,&m,&k);
        road roads[1000009];
        memset(tree,0,sizeof tree);
        for(int i = 0; i < k ; i++){
            scanf("%d%d",&x , &y);
            roads[i].x = x;
            roads[i].y = y;
        }
```

```
        sort(roads, roads+k, compare);

        long long  ans = 0;
        for(int i = 0 ; i < k ; i++){
            ans += (query(m) - query(roads[i].y));
            update(roads[i].y , m);
        }
        printf("Test case %d: %lld\n",l,ans);
    }
    return 0;
}
```

19753102	2017-07-07 20:17:22	Japan	**accepted** edit ideone it	0.13	10M	C++ 4.3.2

BIT-NEXTS-PREVS

```
#include <bits/stdc++.h>

using namespace std;
int N = 20;
void update(int idx) {
  while (idx <= N) {
    cout << idx << ' ';
    idx += (idx & -idx);
  }
}

void query(int i) {
        while (i > 0) {
                cout << i << ' ';
                i -= (i & -i);
        }
}

int main() {
for (int i = 1 ; i <= N ; ++i) {
  cout << "update: nexts of " <<i <<": ";
  update(i);
  cout << '\n';
}

cout << '\n';
for (int i = 1 ; i <= N ; ++i) {
cout << "query: parents of " <<i <<": ";
  query(i);
  cout << '\n';
}
  return 0;
}
```

```
update: nexts of 1: 1 2 4 8 16
update: nexts of 2: 2 4 8 16
update: nexts of 3: 3 4 8 16
update: nexts of 4: 4 8 16
update: nexts of 5: 5 6 8 16
update: nexts of 6: 6 8 16
update: nexts of 7: 7 8 16
update: nexts of 8: 8 16
update: nexts of 9: 9 10 12 16
update: nexts of 10: 10 12 16
update: nexts of 11: 11 12 16
update: nexts of 12: 12 16
update: nexts of 13: 13 14 16
update: nexts of 14: 14 16
update: nexts of 15: 15 16
update: nexts of 16: 16
update: nexts of 17: 17 18 20
update: nexts of 18: 18 20
update: nexts of 19: 19 20
update: nexts of 20: 20

query: parents of 1: 1
query: parents of 2: 2
query: parents of 3: 3 2
query: parents of 4: 4
query: parents of 5: 5 4
query: parents of 6: 6 4
query: parents of 7: 7 6 4
query: parents of 8: 8
query: parents of 9: 9 8
query: parents of 10: 10 8
query: parents of 11: 11 10 8
query: parents of 12: 12 8
query: parents of 13: 13 12 8
query: parents of 14: 14 12 8
query: parents of 15: 15 14 12 8
query: parents of 16: 16
query: parents of 17: 17 16
query: parents of 18: 18 16
query: parents of 19: 19 18 16
query: parents of 20: 20 16
```