**Trước khi học về cây chỉ số nhị phân học sinh tìm hiểu lại kiến thức về các phép toán bit.**

**1. Các phép toán bit cơ bản trong C++**

C++ hỗ trợ nhiều phép toán bitwise cơ bản cho các kiểu dữ liệu số nguyên (int, long, short...). Dưới đây là danh sách các phép toán bit trong C++:

* **Phép toán AND (&)**:
  + Phép toán AND thực hiện phép toán AND giữa các bit của hai toán hạng. Nếu cả hai bit đều là 1, kết quả là 1; ngược lại là 0.

int a = 5; // 0101

int b = 3; // 0011

int result = a & b; // 0001 => result = 1

* **Phép toán OR (|)**:
  + Phép toán OR thực hiện phép toán OR giữa các bit của hai toán hạng. Nếu một trong hai bit là 1, kết quả là 1; nếu không, kết quả là 0.

int a = 5; // 0101

int b = 3; // 0011

int result = a | b; // 0111 => result = 7

* **Phép toán XOR (^)**:
  + Phép toán XOR thực hiện phép toán XOR giữa các bit của hai toán hạng. Nếu các bit khác nhau, kết quả là 1; nếu các bit giống nhau, kết quả là 0.

int a = 5; // 0101

int b = 3; // 0011

int result = a ^ b; // 0110 => result = 6

* **Phép toán NOT (~)**:
  + Phép toán NOT (hay còn gọi là phủ định) đảo ngược tất cả các bit của một số. Các bit 0 trở thành 1 và ngược lại.

int a = 5; // 0101

int result = ~a; // 1010 => result = -6 (vì C++ dùng biểu diễn số âm theo dạng bù 2)

* **Phép toán dịch trái (<<)**: tương đương với **nhân cho 2 mũ k**.(k là số bit dịch
  + Phép toán dịch trái dịch tất cả các bit sang trái một số vị trí nhất định. Các bit mới được đưa vào từ bên phải sẽ là 0.

int a = 5; // 0101

int result = a << 1; // 1010 => result = 10

* **Phép toán dịch phải (>>) (**tương đương với **chia cho 2 mũ k)**
  + Phép toán dịch phải dịch tất cả các bit sang phải một số vị trí nhất định. Các bit mới được đưa vào từ bên trái sẽ là 0 (với số dương).

int a = 5; // 0101

int result = a >> 1; // 0010 => result = 2

**2. Phép toán AND với một bit ( &)**

Kiểm tra một bit cụ thể (bit thứ pos) trong một số nguyên (ví dụ số a) có phải là 1 hay không.

**Công thức:** a & (1 << pos)

|  |
| --- |
| int a = 5; // 0101  int pos = 2; // Kiểm tra bit thứ 2  if (a & (1 << pos)) {  cout << "Bit at position " << pos << " is set (1)." << endl;  } else {  cout << "Bit at position " << pos << " is not set (0)." << endl;  } |

**Giải thích:**  
Sử dụng phép toán dịch trái (1 << pos) để tạo ra một số với chỉ một bit tại vị trí pos là 1, sau đó sử dụng phép toán AND để kiểm tra xem bit tại vị trí đó có phải là 1 không.

3. Đặt bit tại vị trí nhất định (OR)

Đặt một bit cụ thể thành 1 mà không thay đổi các bit khác.

**Công thức:**  
a |= (1 << pos)

|  |
| --- |
| int a = 5; // 0101  int pos = 1; // Đặt bit thứ 1 thành 1  a |= (1 << pos);  cout << "New value of a: " << a << endl; // 0111 => a = 7 |

**Giải thích:**  
Phép toán OR | sẽ đặt bit thứ pos của a thành 1 mà không thay đổi các bit khác.

**3. Xóa bit tại vị trí nhất định (AND với phủ định)**

Xóa một bit tại một vị trí cụ thể (đặt bit thành 0).

**Công thức:**  
a &= ~(1 << pos)

|  |
| --- |
| int a = 5; // 0101  int pos = 2; // Xóa bit thứ 2  a &= ~(1 << pos);  cout << "New value of a: " << a << endl; // 0001 => a = 1 |

**Giải thích:**  
Phép toán phủ định ~ được sử dụng để tạo một số với tất cả các bit là 1 ngoại trừ bit tại vị trí pos, sau đó sử dụng phép toán AND & để xóa bit tại vị trí đó.

**4. Đảo ngược tất cả các bit (NOT)**

Đảo ngược các bit của số nguyên.

**Công thức:**  
~a

|  |
| --- |
| int a = 5; // 0101  int result = ~a; // 1010 => result = -6 (do sử dụng biểu diễn bù 2)  cout << "Result after NOT operation: " << result << endl; |

**5. Ví dụ về ứng dụng của phép toán bit:**

**- Kiểm tra số chẵn hoặc lẻ**:  
Dùng phép toán AND & để kiểm tra xem số có phải là số lẻ hay số chẵn.

|  |
| --- |
| int num = 4;  if (num & 1) {  cout << num << " is odd." << endl;  } else {  cout << num << " is even." << endl;  } |

**- Số bit 1 trong một số nguyên (Hamming Weight)**:  
Đếm số bit 1 trong số nguyên bằng cách sử dụng phép toán AND và dịch.

|  |
| --- |
| int num = 29; // 11101  int count = 0;  while (num) {  count += num & 1;  num >>= 1;  }  cout << "Number of set bits: " << count << endl; |

### **- Tắt bit tại vị trí bất kì**

Tắt bit có nghĩa là gán bit tại vị trí đó bằng 0. Ta thấy x & 0 = 0, nên có thể dùng AND để tắt bit.

Giả sử cần tắt bit K trong số X. Ta thực hiện AND X với số MASK thỏa:

* Bit ở vị trí K bằng 0
* Tất cả các bit còn lại bằng 1

Ví dụ, để tắt bit thứ 0, dùng x & -2.

Kết hợp với phép NOT, ta có thể tạo ra MASK để tắt bit tại vị trí K: X & ~(1 << K)

### **- Tìm bit thấp nhất khác 0**

Đây chính là thao tác cơ bản nhất của [cây Fenwick](https://vietcodes.github.io/algo/fenwick). Để cho ngắn gọn ta kí hiệu “bit thấp nhất khác 0” của x là F(x).

Một số ví dụ:

* F(100) = 4, 100 có biểu diễn nhị phân là 1100100, bit thấp nhất khác 0 là 100 có giá trị là 4.
* F(64) = 64, mở rộng hơn, F(x) = x khi x là lũy thừa của 2.
* F(số lẻ) = 1, do số lẻ luôn có bit thấp nhất = 1.

Công thức: F(x) = x & -x

Phép tính này tận dụng cách biểu diễn bù 2 để tính toán. Để hiểu rõ, bạn đọc nên tính tay trên giấy.

### **Duyệt Tất Cả Các Tập Con Sử Dụng Phép Toán Bitwise**

Trong bài toán duyệt tất cả các tập con của một tập hợp, ta có thể sử dụng số nhị phân để đại diện việc chọn/không chọn một phần tử. Mỗi bit trong một số nguyên có thể đại diện cho một phần tử của tập hợp: nếu bit đó là 1, thì phần tử đó được chọn, ngược lại, nếu bit đó là 0, phần tử đó không được chọn.

Ví dụ:

* Số **13** có biểu diễn nhị phân là 1101. Điều này có nghĩa là tập hợp này chứa các phần tử tại các vị trí **0, 2, 3** (bắt đầu từ 0).
* Các tập con của tập hợp này sẽ được biểu diễn dưới dạng nhị phân như: 1100, 1001, 0101, 0001, ...

Để duyệt tất cả các tập con, ta có thể sử dụng phép toán **AND** kết hợp với việc giảm dần giá trị của k. Mỗi giá trị của k trong quá trình lặp sẽ đại diện cho một tập con của tập hợp ban đầu.

|  |
| --- |
| #include <bits/stdc++.h>using namespace std;int main() {int x = 13; // 1101 trong hệ nhị phân, tương ứng với tập hợp {0, 2, 3}// Duyệt qua tất cả các tập con của xfor (int k = x; k > 0; k = (k - 1) & x) {cout << bitset<4>(k) << '\n'; // In ra biểu diễn nhị phân của mỗi tập con}} |

### Giải thích Mã Nguồn

1. **x = 13**:
   * Biểu diễn nhị phân của x là 1101, tức là tập {0, 2, 3}.
2. **for (int k = x; k > 0; k = (k - 1) & x)**:
   * k = x bắt đầu từ giá trị ban đầu của x, sau đó lặp qua tất cả các giá trị của k là tập con của x.
   * Trong mỗi vòng lặp, k = (k - 1) & x sẽ tạo ra một tập con mới, bằng cách giảm giá trị của k và đảm bảo rằng k luôn là một tập con của x.
3. **bitset<4>(k)**:
   * In ra giá trị k dưới dạng nhị phân với độ dài 4 bit (để dễ nhìn và so sánh).
   * Mỗi giá trị k đại diện cho một tập con của x.

### **Đảo Bit Tại Một Vị Trí Bất Kỳ**

Để đảo bit tại một vị trí cụ thể trong số nguyên, ta có thể sử dụng phép toán **XOR (^)**. Phép toán XOR sẽ đảo bit tại vị trí nếu cần thiết, ví dụ:

* Nếu bit tại vị trí đó là 1, XOR với 1 sẽ biến nó thành 0.
* Nếu bit tại vị trí đó là 0, XOR với 1 sẽ biến nó thành 1.

### Để đảo bit tại vị trí K của số X, ta sử dụng công thức: X ^ (1 << K)

### Ví Dụ Đảo Bit

Giả sử X = 13 (biểu diễn nhị phân là 1101), và ta muốn đảo bit tại vị trí K = 1.

**Bước 1**: Tạo số 1 << K, tức là dịch số 1 sang trái K vị trí.

|  |
| --- |
| 1 << 1 // 0010 |

**Bước 2**: Thực hiện XOR giữa X và 1 << K để đảo bit tại vị trí K.

|  |
| --- |
| X = 13; // 1101  K = 1;  X ^ (1 << K) // 1101 ^ 0010 = 1111 |

**Kết Quả**: Sau khi đảo bit tại vị trí K = 1, ta có kết quả 1111 (tương đương với 15 trong hệ thập phân).

|  |
| --- |
| #include <bits/stdc++.h>  using namespace std;  int main() {  int X = 13; // 1101 trong hệ nhị phân  int K = 1; // Vị trí bit cần đảo  // Đảo bit tại vị trí K  int result = X ^ (1 << K);  // In ra kết quả  cout << "Giá trị ban đầu X: " << bitset<4>(X) << endl; // 1101  cout << "Kết quả sau khi đảo bit tại vị trí " << K << ": " << bitset<4>(result) << endl; // 1111  cout << "Kết quả dưới dạng thập phân: " << result << endl; // 15  } |

**CHUYÊN ĐỀ CÂY CHỈ SỐ NHỊ PHÂN BIT\_FENWICK TREE**

**Fenwick Tree (Binary Index Tree)**

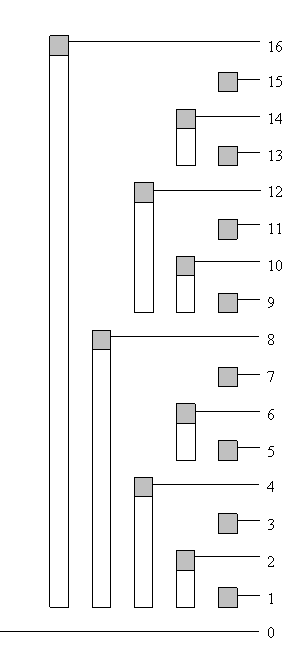
Fenwick Tree, hay còn gọi là cây chỉ số nhị phân (Binary Indexed Tree - BIT), là một cấu trúc dữ liệu tối ưu cho việc cập nhật giá trị một phần tử và tìm tổng, min/max giữa 2 vị trí bất kì trong mảng. Độ phức tạp cho mỗi lần cập nhật, truy xuất là O(logN)) với N là độ dài dãy cần quản lý. Ngoài thao tác tính tổng, tìm min/max thì BIT còn có thể sử dụng được cho nhiều thao thác khác nữa.

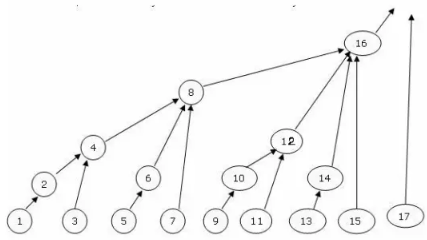
### 2. Ưu nhược điểm:

* Ưu điểm:
  1. Bộ nhớ thấp
  2. Cài đặt đơn giản
  3. Có thể giải được nhiều bài toán về dãy số
  4. Thời gian chạy: O(logn)
* Nhược điểm:

Không tổng quát bằng Segment Tree (Một cấu trúc dữ liệu giải thuật khác). Tất cả những bài giải được bằng Fenwick tree đều có thể giải được bằng Segment Tree. Nhưng chiều ngược lại thì không đúng.

Cây chỉ số nhị phân được xây dựng như sau:





Cấu trúc BIT có bản chất từ ý tưởng cộng dồn và mã hóa các đoạn cộng dồn được quản bằng chỉ số nhị phân.

Bài toán: giả sử có 1 mảng số nguyên gồm n phần tử n=16.

Như vậy biểu diễn nhị phân sẽ gồm 4 bit tương đương với log2(n).

0000, 0001 …🡪1111.

Vấn đề còn lại là làm thế nào để xây dựng được cây BIT như trên? Cách thực hiện là ban đầu khởi tạo các nút của cây BIT là 0. Sau đó ứng với mỗi giá trị am thì cập nhật các nút cha liên quan trong cây. Ví dụ:

    - Cập nhật giá trị a5 -> cần cập nhật các nút t5, nút cha t6, nút cha t8, nút cha t16,….

    - Cập nhật giá trị a9 -> cần cập nhật các nút t9, nút cha t10, nút cha t12, nút cha t16,…

    - Cập nhật giá trị a4 -> cần cập nhật các nút t4, nút cha t8, nút cha t16,…

Tổng quát với m bất kỳ, biểu diễn m thành dạng nhị phân, nếu cộng 1 vào bít bên phải nhất của m thì ta được nút cha của m.

*Ví dụ*, m = 5 có biểu diễn nhị phân là 101:

    1) 101 -> cập nhật nút 5

    2) Cộng 1 vào bít phải nhất thành 0110 -> cập nhật nút 6

    3) Cộng 1 vào bít phải nhất thành 1000 -> cập nhật nút 8

    4) Cộng 1 vào bít phải nhất thành 10000 -> cập nhật nút 16.

Thao tác cập nhật các nút từ con đến cha như trên được gọi là updateBIT. Ta cũng có một công thức rất đơn giản để cộng 1 vào bít 1 bên phải nhất dùng phép toán AND. Thủ tục updateBIT như sau:

|  |  |
| --- | --- |
| void updateBIT(int m, int value)  {    for(; m<= n; **m += m & -m**)     {          t[m] += value;  } } |  |

Độ phức tạp của updateBIT là O(log2n).

|  |  |
| --- | --- |
| #include <bits/stdc++.h>  using namespace std;  int n,m, a[100002];  int f[10002];  int main()  {  cin>>n>>m;  for(;m<=n;m+=m&-m)  {  cout<<m<<" ";  }  } | 16 1  1 2 4 8 16  16 3  3 3 8 16  Đây là bước cập nhật các nút quản lý của nút 1 và nút 3 giới hạn đến n. |

Ngược giả sử tại nút 13 muốn tính giá trị tại nút 13 thì chỉ cần tính:

Giá trị tại nút 13, 12 và 8

Sum[13]= f[13]+f[12]+f[8]

Theo xây dựng cây giá trị tại nút 13 sẽ là nút 13

Tại nút 12 sẽ quản lý đoạn [9,10,11,12] và 8 sẽ quản lý đoạn từ [1🡪8]

|  |
| --- |
| *Ví dụ*, m = 13 có biểu diễn nhị phân là 1101:       1) 1101 -> truy xuất nút 13       2) Xóa bít 1 bên phải nhất còn 1100 -> truy xuất nút 12       3) Xóa bít 1 bên phải nhất còn 1000 -> truy xuất nút 8       4) Xóa bít 1 bên phải nhất và dừng.  Thao tác truy xuất các nút như trên được gọi là getBIT. May mắn là ta có một công thức rất đơn giản để xóa bít 1 bên phải dùng phép toán AND. Thủ tục getBIT như sau: |

|  |  |
| --- | --- |
| #include <bits/stdc++.h>  using namespace std;  int n,m, a[100002];  int f[10002];  int main()  {  cin>>m;  for(;m>=1;m-=m&-m)  {  cout<<m<<" ";  }  } |  |

Ví dụ mảng A có 10 phần tử và cây BIT tương ứng:

| **i** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A | 3 | 2 | 1 | 5 | 3 | 6 | 2 | 0 | 7 | 1 |
| BIT | 3 | 5 | 1 | 11 | 3 | 9 | 2 | 22 | 7 | 8 |
| Nút do BIT quản lý | 1 | 1..2 | 3 | 1..4 | 5 | 5..6 | 7 | 1..8 | 9 | 9..10 |

Ví dụ :

51 🡪 110011 🡪 25+ 24+ 21+20 = 32+16+2+1

Tức là muốn tính bit[51]

Chỉ cần cập nhật: [1🡪32] + [33🡪 48] +[49🡪50]+ [51]

Bây giờ xem nút 51 quản lý những nút nào

Bằng câu lệnh: m=m-(m&-m)

Đầu tiên 51 quản lý chính nó.

Sau đó 51-1=50 // (m&-m) trả về vị trí đầu tiên xuất hiện bít 1.

50-2=48

48- 16 = 32

32 – 32 =0;

**Bài tập cơ bản 1:**

Cho mảng a gồm n phần tử, q truy vấn:

Mỗi truy vấn bao gồm 1 số t:

Nếu t=1, nhập x, val🡪 gán ax=v.

Nếu t=2, nhập x; tính tổng các số từ a1 đến ax.

Yêu cầu: In ra giá trị khi gặp truy vấn 2.

Input: Số nguyên n, t là số phần tử mảng, số lượng truy vấn n, q<= 200000

Dòng tiếp theo là n số nguyên ai <= 109.

T dòng tiếp theo bao gồm 1 số t ( t=1 hoặc t=2).

Nếu t=1 nhập x, v;

Nếu t=2 nhâp 1 số x (x<=n, v<=109)

In kết quả mỗi truy vấn 2 trên 1 dòng.

|  |  |
| --- | --- |
|  |  |
| 5 3  1 2 3 4 5  1 3 5  1 4 6  2 4 | 14 |

|  |
| --- |
| #include <bits/stdc++.h>  using namespace std;  long long BIT[100005];  int n,t,a[100005];  int get(int x)  {  int s=0;  while(x>0)  {  s+=BIT[x];  x-=x&-x;  }  return s;  }  void update(int x, int value)  {  while(x<=n)  {  BIT[x]+=value;  x+=x&-x;  }  }  main()  {  ios\_base::sync\_with\_stdio(0);  cin.tie(0);  cout.tie(0);  cin>>n>>t;  for(int i=1; i<=n; i++)  {  cin>>a[i];  update(i,a[i]);  }  while(t--)  {  int u;  cin>>u;  if(u==1)  {  int x,val;  cin>>x>>val;  update(x,val-a[x]);  a[x]=val;  }  else  {  int y;  cin>>y;  cout<<get(y)<<endl;  }  }  } |

**Bài tập vận dụng:**

**Bài 1. Số nghịch thế (INVERS.\*) – Nguồn thầy Nguyễn Thanh Tùng**

Xét dãy số nguyên ***A*** = (***a1***, ***a2***, . . ., ***an,***)(1 ≤ ***n*** ≤ 500 000). Các số trong dãy ***A*** khác nhau từng đôi một và nhận giá trị trong phạm vi từ 1 đến ***n***. Như vậy dãy ***A*** là một hoán vị các số từ 1 đến ***n***. Cặp số (***ai***, ***aj***)trong dãy ***A*** được gọi là một nghịch thế, nếu ***i*** < ***j*** và ***ai*** > ***aj***.  
***Yêu cầu***: Cho ***n*** và hoán vị ***A***. Hãy xác định số nghịch thế.  
***Dữ liệu***: Vào từ file văn bản INVERS.INP:

* Dòng đầu tiên chứa số nguyên ***n***,  
  Dòng thứ 2 chứa n số nguyên xác định hoán vị ***A***.

***Kết quả***: Đưa ra file văn bản INVERS.OUT một số nguyên – số lượng nghịch thế.  
***Ví dụ***:

|  |  |
| --- | --- |
| **INVERS.INP** | **INVERS.OUT** |
| 5  2 4 3 5 1 | 5 |

**Bài 2. Xếp hàng ( XEPHANG.\*)**

Hàng ngày khi lấy sữa, N con bò của bác John (1 ≤ N ≤ 50000) luôn xếp hàng theo thứ tự không đổi. Một hôm bác John quyết định tổ chức một trò chơi cho một số con bò. Để đơn giản, bác John sẽ chọn ra một đoạn liên tiếp các con bò để tham dự trò chơi. Tuy nhiên để trò chơi diễn ra vui vẻ, các con bò phải không quá chênh lệch về chiều cao.

Bác John đã chuẩn bị một danh sách gồm Q (1 ≤ Q ≤ 200000) đoạn các con bò và chiều cao của chúng (trong phạm vi [1, 1000000]). Với mỗi đoạn, bác John muốn xác định chênh lệch chiều cao giữa con bò thấp nhất và cao nhất. Bạn hãy giúp bác John thực hiện công việc này!

**Dữ liệu vào:**

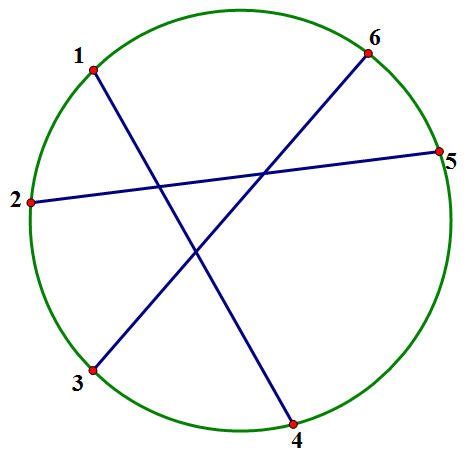
* Dòng đầu tiên chứa 2 số nguyên N và Q.
* Dòng thứ i trong số N dòng sau chứa 1 số nguyên duy nhất, là độ cao của con bò thứ i.
* Dòng thứ i trong số Q trong tiếp theo chứa 2 số nguyên A, B (1 ≤ A ≤ B ≤ N), cho biết đoạn các con bò từ A đến B.

**Kết quả:**

Gồm Q dòng, mỗi dòng chứa 1 số nguyên, là chênh lệch chiều cao giữa con bò thấp nhất và cao nhất thuộc đoạn tương ứng.

|  |  |
| --- | --- |
| **XEPHANG.inp** | **XEPHANG.out** |
| 6 3  1  7  3  4  2  5  1 5  4 6  2 2 | 6  3  0 |

**Bài 3. Dây cung ( DAYCUNG.\*) – Nguồn sưu tầm**

****Có 2n điểm khác nhau được đánh dấu trên đường tròn. Các điểm được đánh số từ 1 tới 2n theo chiều ngược kim đồng hồ (1 ≤ n ≤ 100 000).

Vẽ n dây cung, dây thứ i nối hai điểm ai và bi. Mỗi điểm đã cho chỉ thuộc đúng một dây cung.

***Yêu cầu***: Cho n và các số ai, bi (1 ≤ i ≤ n). Hãy xác định số cặp dây cung giao nhau.

**Dữ liệu vào**: Đọc từ file **DAYCUNG.INP**

* Dòng đầu tiên chứa số nguyên n.
* Dòng thứ i trong n dòng sau chứa hai số nguyên ai và bi.

**Dữ liệu ra:** Ghi ra file **DAYCUNG.OUT**

* Một số nguyên là số cặp dây cung giao nhau.

Ví dụ:

|  |  |
| --- | --- |
| **DAYCUNG.INP** | **DAYCUNG.OUT** |
| 3  1 4  2 5  3 6 | 3 |

**Giới hạn:**

* 50% số test có 1 ≤ N 1000.
* 50% số test với các trường hợp còn lại.