Chuyên đề:

Các cấu trúc dữ liệu đặc biệt

Chỉ cần qua câu nói "Algorithms+Data Structures = Program" của Niklaus Wirth ta đã có thể thấy được tầm quan trọng của các loại cấu trúc dữ liệu [data structures] trong giải các bài toán tin. Ứng dụng 1 cách thuần thục hiệu quả các loại cấu trúc sẽ đem đến những thuận lợi vô cùng lớn cho các lập trình viên. Ngoài những cấu trúc dữ liệu chuẩn, quen thuộc như array, record, queue,... còn có 1 số cấu trúc dữ liệu khác có hiệu quả đặc biệt trong 1 số dạng bài tập. Mặc dù vậy, tài liệu tiếng việt về những cấu trúc này lại khá ít, đặc biệt là Interval Tree, Binary Indexed Tree và Range minimum Query. Trong chuyên đề này sẽ đề cập tới 1 số loại cấu trúc thường xuyên được sử dụng: Interval tree, Binary Indexed Tree, Heap, Range Minimum Query và Disjoint set. Để giúp bạn hiểu rõ các cấu trúc đó, sẽ có thêm 1 số ví dụ điển hình cho những ứng dụng khác nhau của chúng.

I. Interval Tree.

Interval Tree là 1 cấu trúc vô cùng hữu dụng, được sử dụng rất nhiều trong các bài toán về dãy số. Ngoài ra Interval Tree còn được sử dụng trong 1 số bài toán hình học. Có thể nói nếu nắm rõ Interval Tree bạn đã làm được 1 nửa số bài toán về dãy số rồi đấy!.

Xin nói thêm thực ra Interval Tree tên gọi chính xác là Segment Tree nhưng cái tên Interval Tree được sử dụng nhiều hơn ở Việt Nam <các nước khác thì ... không rõ lắm>. Nếu tìm trong "Introduction to Algorithms 2nd Edition" thì bạn sẽ thấy 1 cấu trúc mang tên Interval tree nhưng với nội dung khác so với những gì sẽ được trình bày ở dưới đây.

Ta sẽ xem xét 1 bài toán đơn giản sau để hiểu thế nào là cây Interval Tree:

Bài toán: Cho 1 dãy số gồm N phần tử (N<=10000) ban đầu đều mang giá trị 0. Có 2 loại thao tác cơ bản:

1. INC i gtr : tăng giá trị phần tử thứ i lên gtr đơn vị.
2. GET L R : tìm và trả về tổng giá trị của các phần tử từ L tới R.

Trong file input có M<=100000 thao tác. Yêu cầu tương ứng với mỗi thao tác GET đưa ra kết quả tương ứng trong file output. Dữ liệu đảm bảo các kết quả trong phạm vi longint.

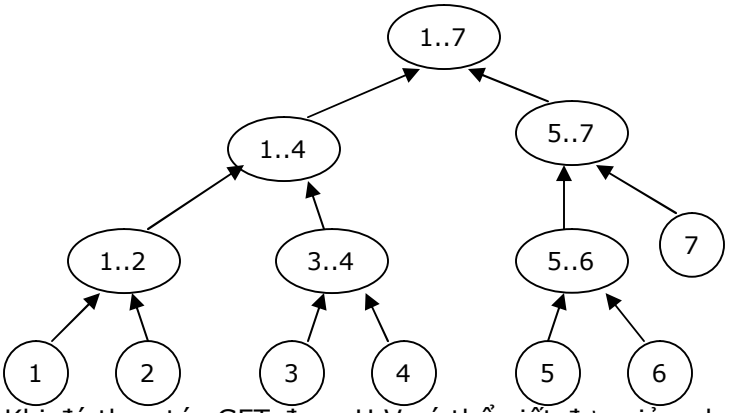
Thuật toán đơn giản nhất cho bài toán này là làm thô thiển: với mỗi thao tác INC ta tăng giá trị của A[i] và với mỗi thao tác GET tính lại tổng các số trong đoạn từ L tới R. Độ phức tạp thuật toán là O(M*N) không thể chạy được với những test lớn.

Vậy làm cách nào để cải tiến thuật toán? Ta có thể dùng Interval Tree để làm giảm độ phức tạp của phép lấy tổng. Nếu làm như trên, mỗi thao tác GET sẽ thực hiện trong O(N), nếu dùng Interval Tree thì độ phức tạp chỉ còn là O(logN), bằng cách tính trước 1 số đoạn nhỏ trong đoạn [L,R] cần tính và khi tính tổng đoạn [L,R] chỉ cần tính tổng các đoạn nhỏ nằm trong nó.

Cấu trúc cây Interval được sử dụng có thể mô tả như sau:

- Gốc của cây là nút lưu tổng (tức là quản lý) các đoạn trong khoảng từ 1..N
- Xét 1 nút bất kì lưu tổng đoạn từ L..R
  + Nếu L=R nút này không có con
  + Nếu L<>R, nút này có 2 con: con trái lưu tổng đoạn từ L tới Mid và con phải lưu tổng đoạn từ Mid+1 tới R, trong đó Mid=(L+R) div 2.

VD: nếu 1 cây interval tree của 1 dãy có N=7 phần tử thì đồ thị miêu tả cây này sẽ có dạng: (số ghi tại mỗi nút là đoạn phần tử nó quản lý)



Khi đó thao tác GET đoạn U,V có thể viết đơn giản như sau:
GET(U,V,L,R) {lấy tổng đoạn U..V, đang xét đoạn L..R}
1. if (V<L) or (U>R) --> exit {ngoài đoạn U..V}
2. if (U>L) and (V<R) --> result+=A[L..R]; exit {thuộc hoàn toàn trong đoạn U..V}
3. result+=GET(U,V,L,mid)+GET(U,V,mid+1..R); {đoạn đang xét giao với đoạn cần lấy tổng}
Và gọi GET(U,V,1,n) với result khởi tạo bằng 0.
Dễ thấy số lần thực hiện đệ quy nhỏ hơn O(2*logN) vì thủ tục GET này chỉ được gọi tiếp khi đoạn L..R giao và không thuộc đoạn U..V. Như vậy thao tác GET thay vì thực hiện trong O(N) nay đã có thể thực hiện trong O(logN). Còn thao tác INC thì sao? Rõ ràng thao tác INC không chỉ đơn giản là cập nhật lại phần tử thứ I như trước mà ta còn phải điều chỉnh cả cây Interval Tree sao cho đúng với mô tả. Để update lại cây Interval Tree với mỗi thao tác tăng giá trị phần tử thứ I ta phải tăng mỗi nút của cây mà quản lý I lên 1 giá trị gtr. Hàm INC được viết lại:
INC(i,gtr,L,R) {xét nút L..R, cần tăng I lên gtr đơn vị}
1. if (L>i) or (R<i) --> exit {khoảng đang xét không chứa i}
2. A[L..R]+=gtr {nút này có chứa I}
3. INC(i,gtr,L,mid)
4. INC(i,gtr,mid+1,R)
Độ phức tạp của thao tác INC cũng không vượt quá O(2*logN) vì độ cao của cây interval luôn nhỏ hơn logN.
Như vậy độ phức tạp của thao tác INC mặc dù tăng từ O(1) lên O(logN) nhưng độ phức tạp chung của bài toán chỉ còn lại là O(MlogN) nhanh hơn hẳn so với thuật toán thô sơ ban đầu.
(Lưu ý: đây chỉ là bài ví dụ, trên thực tế có những cách nhanh hơn để xử lý bài toán này mà không dùng tới interval tree).
Qua ví dụ trên ta cũng có thể hiểu qua phần nào về cấu trúc và ý nghĩa sử dụng của Interval tree: Gốc là nút lưu toàn bộ thông tin (mà trong ví dụ là tổng) từ 1 tới N, từ gốc thông tin 1 nút được chia nhỏ ra quản lý ở 2 nút con trái và phải cho tới khi mỗi nút chỉ lưu thông tin của 1 phần tử. Lợi ích trong phương pháp sử dụng interval tree là với 1 số đoạn con ta có thể lấy trực tiếp được thông tin trong đoạn con đó mà không phải đi lấy thông tin trong từng phần tử nhỏ trong đoạn, việc này giúp giảm độ phức tạp trong các thao tác từ O(N) xuống O(logN).
Tư tưởng của Interval tree là dùng "chia để trị": "chia" đoạn lớn thành các đoạn nhỏ hơn để có thể "trị" nhanh chóng.
1 câu hỏi khác được đặt ra là lưu trữ interval tree trong thực tế như thế nào, vì ta không thể hầu như không thể bỏ ra N^2 đoạn để lưu các đoạn từ L..R được, quá tốn bộ nhớ với N lớn. Câu trả lời là ta sẽ dùng 1 mảng 1 chiều để lưu trữ interval tree:
Data for Interval Tree
1. Root lưu đoạn 1..N, được lưu trữ ở A[1].
2. Nếu Node I lưu đoạn [L..R] và L<R {đoạn L..R được lưu trữ ở A[i]}

3. Thì Node 2*I lưu đoạn [L..mid] và Node 2*I+1 lưu đoạn [mid+1..R].
Độ cao của interval tree luôn nhỏ hơn hoặc bằng logN. Như vậy bộ nhớ dùng cho interval tree là O(2^(logN+1). Trong thực tế có thể khai báo mảng O(N*3) hoặc O(N*4) là đủ. (Tất nhiên cũng có thể dùng Linklist – danh sách động để lưu interval tree nhưng cách này tốn bộ nhớ hơn và không tiện bằng, không hay được dùng).
Nhược điểm của cách lưu này là ta không thể biết được đoạn [L..R] có được lưu trọn trong 1 nút không và nếu có thì nút đó là nút nào mà buộc phải lặp lại 1 quá trình với độ phức tạp O(logN) từ gốc tới đoạn [L..R].
Các thông tin được lưu trong 1 node của interval tree là thông tin tổng hợp của đoạn mà nó quản lý, bởi vậy thông tin này phải là dạng tích luỹ được, ví dụ như tổng, hiệu, min hoặc max,...
Từ sau đây ta gọi chung các thao tác sửa cây Interval là các thao tác UPDATE, các thao tác lấy thông tin từ cây là thao tác GET.
Ứng dụng của cây Interval tree đa dạng, phong phú vô cùng. Sau đây ta sẽ tìm hiểu một số ứng dụng cơ bản và hay gặp nhất. Mỗi ví dụ sẽ mô tả 1 cách sử dụng interval tree tương đối khác nhau và thường gặp trong giải toán.

Ví dụ 1. Các bài toán cơ bản ứng dụng Interval Tree:
a. Cho dãy số, có 1 số yêu cầu thuộc 2 loại thay đổi (tăng/gán lại) giá trị 1 phần tử hoặc tìm min, max các đoạn liên tiếp của dãy số: mỗi nút interval tree sẽ lưu giá trị min/max các phần tử nó quản lý.
b. Cho dãy số, có 1 số yêu cầu gán thuộc 2 loại lại giá trị của 1 phần tử hoặc tìm tổng 1 số phần tử liên tiếp của dãy. Bài toán tương tự như ví dụ.

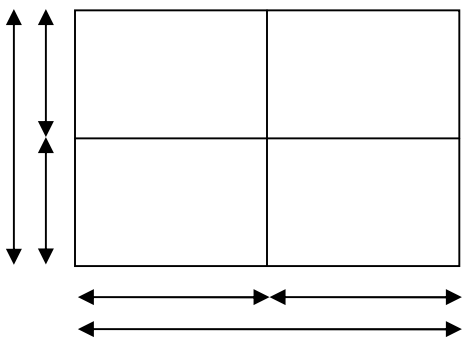
Ví dụ 2: POSTERS – AMPPZ 2001.
Tóm tắt đề bài: Có N tấm poster chiều cao 1. Theo thứ tự các tấm poster được dán lên 1 đoạn tường cũng có chiều cao 1. Đoạn tường được xây bởi các viên gạch 1*1, đánh số từ trái sang phải bắt đầu từ 1. Các tấm poster sẽ phủ 1 đoạn liên tiếp từ viên gạch Li tới viên gạch Ri, tấm poster được dán sau sẽ phủ lên tấm poster được dán trước. Vì vậy, sau khi dán xong cả N tấm poster thì có thể có những tấm poster không thể được nhìn thấy.
Yêu cầu: Đếm số loại poster khác nhau có thể nhìn thấy được từ ngoài vào.
Input: Dòng đầu ghi N là số tấm poster. Trong N dòng tiếp theo mỗi dòng chứa 2 số Li và Ri thể hiện đầu trái và đầu phải của tấm poster thứ i.
Output: Duy nhất 1 dòng ghi số loại poster có thể nhìn thấy được.
Giới hạn: N<=40000. Li,Ri<=10^9.
Hướng dẫn:
Bài toán có thể phát biểu 1 cách dễ hiểu như sau: Cho dãy số M phần tử, có 1 số thao tác tô màu các phần tử của dãy số. Sau khi kết thúc chuỗi thao tác đếm số màu khác nhau của dãy số trên. Với M nhỏ, ta chỉ cần lưu lại được màu của các phần tử sau đó xem có bao nhiêu màu khác nhau là được. Nhưng nếu xét trong bài toán POSTERS này, thì M của chúng ta sẽ có thể lên tới giá trị 10^9. Do đó, ta phải làm nhỏ lại giá trị này. Bằng cách nào? Nhận xét với 2 ô mà giữa chúng không có đầu mút của tấm poster nào thì chắc chắn màu sắc của chúng giống nhau. Từ đó ta thực hiện trộn tất cả các đầu mút của các đoạn, sắp xếp tăng dần chúng. Thay vì phải xét tất cả các ô (có thể lên tới 10^9 ô) ta chỉ cần xét các ô là đầu mút của các đoạn, số lượng này chỉ khoảng 80000 số, hoàn toàn có thể lưu trữ được. Phương pháp ta vừa áp dụng còn được gọi là phương pháp "Rời rạc hoá", ứng dụng hiệu quả nhiều trong các bài toán khác nhau, nhất là khi sử dụng các cấu trúc dữ liệu đặc biệt. Ý nghĩa chủ yếu là với 1 đoạn lớn các phần tử giống hệt nhau, không cần xét mọi phần tử mà chỉ xét 1 phần tử đại diện. Sau đây các bạn sẽ còn gặp nhiều bài toán sử dụng phương pháp này.
Trở lại với bài toán của chúng ta, bây giờ phải sửa đổi màu các phần tử trong 1 đoạn

liên tiếp. Với giới hạn M còn 80000 ta vẫn không thể làm thô được, mà sẽ dùng interval tree. Vì cuối cùng cần màu của mỗi phần tử nên cây interval được xây dựng phải bảo đảm điều kiện lấy được màu của các phần tử. Có 2 hướng lưu trữ cây interval như sau:
1/ Tại mỗi nút lưu màu chung của các phần tử nó quản lý, khởi tạo là màu 0. Chính xác hơn là mỗi nút lưu màu cuối cùng mà nó được sửa, kèm theo thời gian nó được sửa thành màu đó. Quá trình sửa màu vẫn diễn ra bình thường nhưng kết hợp thêm cập nhật thời gian. Màu của 1 phần tử khi đó là màu của nút quản lý nó mà màu được cập nhật muộn nhất. Dễ thấy giá trị đó đúng là màu của phần tử đang xét. Để lấy màu ta chỉ cần đi từ gốc tới nút chứa duy nhất phần tử đó và chọn màu có thời gian lớn nhất.
2/ Cây Interval lưu không chính xác màu của các nút mà chỉ lưu 1 cách gần đúng. 1 nút lưu màu nếu đó là màu chung của tất cả các phần tử nó quản lý, ngược lại lưu giá trị (-1). Ta sẽ kết hợp quá trình sửa đúng lại màu ch o các phần tử vào trong quá trình cập nhật và lấy giá trị các phần tử. Trong quá trình cập nhật, xét tới nút nào thuộc trong đoạn được tô mới màu thì gán luôn giá trị nút đó bằng màu mới và kết thúc (tương tự như bình thường), những nút cha của nút này được gán giá trị -1 (do màu các nút con của nó không còn giống nhau). Trong quá trình lấy giá trị phần tử, nếu 1 nút cha mang giá trị dương thì nút con sẽ mang giá trị của nút cha thay vì giá trị hiện thời của nó, cập nhật lại màu cần diễn ra trước khi xét tới các nút con của 1 nút.
2 cách lưu trên đều khá đơn giản và dễ hiểu (theo tôi cách đầu tiên dễ hiểu và dễ cài đặt hơn còn cách thứ 2 cần hiểu rõ bản chất và tư duy mạch lạc, nếu không sẽ dễ nhầm lẫn giá trị các nút, nhưng nếu cài tốt sẽ nhanh và đỡ tốn bộ nhớ hơn cách đầu). Bạn nên thử cài lại bài toán theo cả 2 cách đã nêu và chọn cách phù hợp nhất cho mình. Tương tự hãy ứng dụng cây interval vào trường hợp tăng/giảm giá trị và tính tổng 1 số đoạn liên tiếp của dãy số.

Ví dụ 3. MARS Map – Baltic OI 2001.
Trên mặt phẳng toạ độ có N hình chữ nhật, có toạ độ các đỉnh trong khoảng từ 0 tới 30000. Tính diện tích phần mặt phẳng mỗi điểm bị phủ bởi ít nhất 1 hình chữ nhật.
Input: Dòng đầu tiên ghi số N, trong N dòng tiếp theo mỗi dòng ghi 2 cặp số lần lượt là toạ độ điểm trái dưới và phải trên.
Output: Duy nhất 1 số là tổng diện tích phần mặt phẳng bị phủ bởi ít nhất 1 hình chữ nhật.
Giới hạn: N<=10000.
Hướng dẫn:
- Vì các toạ độ đều nguyên, nếu ta chia mặt phẳng thành lưới các ô vuông thì diện tích phần bị phủ bởi các HCN chính là số ô vuông thuộc ít nhất 1 HCN. Như vậy ta chỉ cần đếm với mỗi cột dọc rộng 1 đơn vị có bao nhiêu ô vuông như vậy là được.
- Số ô bị phủ mỗi cột chỉ thay đổi khi các HCN phủ nó thay đổi. Do đó nếu giữa 2 cột i,i+1 không có sự thay đổi về các HCN phủ lên chúng thì số ô vuông bị phủ ở 2 cột này là bằng nhau. Sự thay đổi này chỉ có khi có 1 cạnh của 1 HCN hoàn toàn thuộc trên đường thẳng đứng giữa 2 cột trên.
Từ 2 nhận xét trên ta đi tới thuật toán sau:
- B1: sắp xếp chỉ số vị trí các cạnh thẳng đứng của các HCN theo chiều tăng dần, những cột thuộc giữa 2 chỉ số liên tiếp sẽ có số ô bị phủ bằng nhau, ta chỉ cần đếm lượng này rồi nhân với số lượng cột là được. Do đó chỉ xét 1 cột ngay sau vị trí 1 cạnh là đủ.
- B2: xét các cột từ trái sang phải, nếu lần đầu tiên gặp 1 HCN (gặp cạnh dọc trái của nó) thì thêm đoạn mà nó phủ ở cột tương ứng, nếu đó là cạnh dọc phải của HCN thì loại bỏ đoạn mà nó phủ. Mỗi lần xét 1 cột đếm số lượng ô bị phủ của cột đó.

Ta dùng interval tree cho quá trình này. Bài toán có thể được phát biểu lại như sau: Cho 1 dãy số có N số, có 1 số thao tác là tăng hoặc giảm 1 số phần tử liên tiếp của dãy lên 1 đơn vị, sau mỗi thao tác hỏi dãy số có bao nhiêu số lớn hơn 0. Với giá trị max toạ độ = 30000 thì giá trị N trên có thể lên tới 30000, nếu giá trị này lớn hơn sẽ rất khó khăn trong lưu trữ. Nhưng ta cũng có thể áp dụng phương pháp rời rạc hoá các đoạn liên tiếp giống nhau. Khi đó N lớn nhất chỉ bằng số HCN, tức là 10000 mà thôi, bài toán lúc này khác 1 chút: mỗi phần tử kèm 1 hằng giá trị, tính tổng hằng giá trị các phần tử lớn hơn 0.
Với cách phát biểu này bài toán đã trở nên gần gũi hơn và dễ dàng giải quyết hơn rất nhiều. Chỉ cần lưu kèm mỗi nút cây interval là số lượng phần tử dương nó quản lý. Phần còn lại của bài toán xin nhường cho các bạn tự giải.

Dạng mở rộng của interval tree:
Ta đã thấy được sức mạnh của Interval tree trong xử lý bài toán dãy số. Vậy nếu với 1 bảng số thì sao? Nếu coi dãy số là 1 đoạn thẳng (1 chiều) thì bảng số có thể coi như 1 HCN (2 chiều), có sự mở rộng thêm 1 chiều nữa so với dãy số. Như vậy thì hoàn toàn có thể dùng Interval Tree theo 1 cách nào đó để xử lý các bảng số. Cây Interval Tree khi đó thường được gọi là Interval Tree 2D – Cây interval tree 2 chiều. Nếu như Interval Tree chỉ có 1 cách biểu diễn thông dụng và được dùng (tới 99.9% các bài toán dùng cấu trúc mô tả ở trên) thì lại có tới 2 cách hoàn toàn khác nhau để hiểu và biểu diễn Interval Tree 2D. Vậy thế nào là Interval Tree 2D? Xét ví dụ sau:
Ví dụ 4: MATSUM - Al-Khawarizm 2006
Cho ma trận N*N. Ban đầu tất cả các ô của ma trận đều mang giá trị 0. Các dòng đánh số từ 1 tới N từ trên xuống dưới, các cột được đánh số từ 1 tới N từ trái qua phải. Có 1 trình xử lý gồm 3 thao tác chính trên ma trận:
1. SET x y num : gán giá trị của ô (x,y) giá trị num
2. SUM x1 y1 x2 y2 : Tính và in ra tổng giá trị các ô trong HCN ô trái dưới (x1,y1) và phải trên (x2,y2) (x1<=x2, y1<=y2).
3. END : kết thúc chương trình.
Yêu cầu: viết chương trình đọc vào các lệnh của trình xử lý, tính và đưa ra kết quả của các thao tác SUM.
Giới hạn: N<=1024.
Định nghĩa HCN (x1,x2,y1,y2) là HCN giới hạn bởi 2 hàng x1,x2 và 2 cột y1,y2
Cách 1: Quản lý song song cả 2 chiều:
Với Interval tree thì các đoạn được chia đôi chia đôi dần. Sử dụng tư tưởng này trong Interval Tree 2D thì ta chia đôi theo cả 2D-2direction hàng và cột. Mỗi nút interval tree sẽ quản lý 1 bảng HCN nhỏ trong bảng HCN ban đầu và được chia thành 4 nút con. VD: 1 hình chữ nhật được chia làm 4 hình nhỏ hơn:



Hay nói cách khác 1 nút (x1,x2,y1,y2) có thể có tối đa 4 nút con là (x1,mx,y1,my), (x1,mx,my+1,y1), (mx+1,x2,y1,my), (mx+1,x2,my+1,y2) v ới mx=(x1+x2)/2;my=(y1+y2)/2.
Cây này hoàn toàn tương tự Interval tree, ta chỉ cần quản lý theo 2 chiều, có thể

dùng mảng 1 chiều hoặc 2 chiều để quản lý tuỳ ý.
Áp dụng vào bài toán trên ta lưu tại mỗi nút là tổng giá trị các ô mà HCN đó quản lý. Các hàm GET và UPDATE có thể viết hoàn toàn tương tự hàm với Interval tree, chỉ khác ở điểm từ 1 nút sẽ gọi tới 4 nút con thay vì 2. Độ phức tạp thuật toán cho các thao tác trở thành O(LogM*logN) chứ không phải là O(logN) nữa.
Cách 2: Quản lý lần lượt theo từng chiều:
Với bảng số M*N ta dùng M interval tree quản lý M hàng riêng rẽ (lớp cây T1). Tại nút (i,j) của cây lưu hàng K sẽ lưu tổng các ô từ i tới j của hàng K. Vì mỗi hàng đều có N cột nên số nút ở mỗi cây con này là bằng nhau. Giả sử có P nút con trong mỗi cây con này. Ta sử dụng lớp cây T2 gồm P cây interval nữa, mỗi cây sẽ quản lý M nút: cây thứ P sẽ quản lý nút thứ P của M cây interval trước đó. Vậy giá trị các ô trong HCN sẽ được truy xuất như thế nào? Với 1 HCN (xL,xR,yL,yR) thì đầu tiên ta tìm các nút thuộc đoạn (yL,yR) thuộc lớp cây T1. Với mỗi nút đó truy xuất dữ liệu ở cây tương ứng thuộc T2 và trong đoạn từ xL tới xR. Quá trình UPDATE dữ liệu cũng tương tự. Độ phức tạp thuật toán dạng này cũng là O(logM*logN) cho mỗi thao tác UPDATE và GET.
2 cách biểu diễn trên khác nhau nhưng có cùng độ phức tạp khi xử lý.
Yêu cầu tự viết các chương trình mô tả 2 dạng của cây Interval Tree 2D.

Bài tập tự giải:

1. CUTSEQ – Marathon 06-07.

Cho số nguyên N và một dãy số nguyên a_1, a_2, …, a_N. Nhiệm vụ của bạn là phải cắt dãy số trên thành một số dãy số (giữ nguyên thứ tự) thỏa mãn:
- Tổng của mỗi dãy số không lớn hơn số nguyên M.
- Tổng của các số lớn nhất trong các dãy trên là nhỏ nhất.

Input:
Dòng đầu gồm 2 số nguyên N và M.
Dòng thứ hai gồm N số nguyên của dãy a_1, a_2, …, a_N.
Output:
Gồm một số duy nhất là tổng của các số lớn nhất trong các dãy số trên. Nếu không có cách cắt nào thỏa mãn hai điều kiện trên, in ra -1.
Giới hạn:
$-1 \le N \le 100000$.
$-0 \le a_i \le 10^6$.
$-M < 2^{63}$.

2. The BUS – POI 2004.

Tóm tắt đề bài: Cho lưới ô vuông M*N. Tại K nút (giao của hàng và cột) của lưới có 1 giá trị GT>0, các nút khác giá trị bằng 0. 1 đường đi từ ô (1,1) tới ô (M,N) của lưới là đường đi thoả mãn các điều kiện sau:
- Đi theo các cạnh của lưới ô vuông, không đi theo các đường chéo.
- Chỉ có thể đi từ nút (i,j) tới nút (i+1,j) hoặc nút (i,j+1).

Giá trị của đường đi là tổng giá trị của các nút thuộc đường đi. Tìm đường đi có giá trị lớn nhất và đưa ra file output giá trị này.
Input: dòng đầu tiên ghi 3 số nguyên M N K ý nghĩa trên. K dòng tiếp theo mỗi dòng ghi 3 số X Y GT ý nghĩa là nút (X,Y) có giá trị GT.
Output: 1 dòng duy nhất ghi kết quả tìm được
Giới hạn:
-1<=M,N<=10^9.
-K<=10^5
-Kết quả trong phạm vi longint.

3. POINTS and RECTANGES

Trong mặt phẳng toạ độ cho N hình chữ nhật và M điểm. 1 điểm được gọi là thuộc 1 HCN nếu như điểm đó nằm trong phần mặt phẳng giới hạn bởi HCN đó.

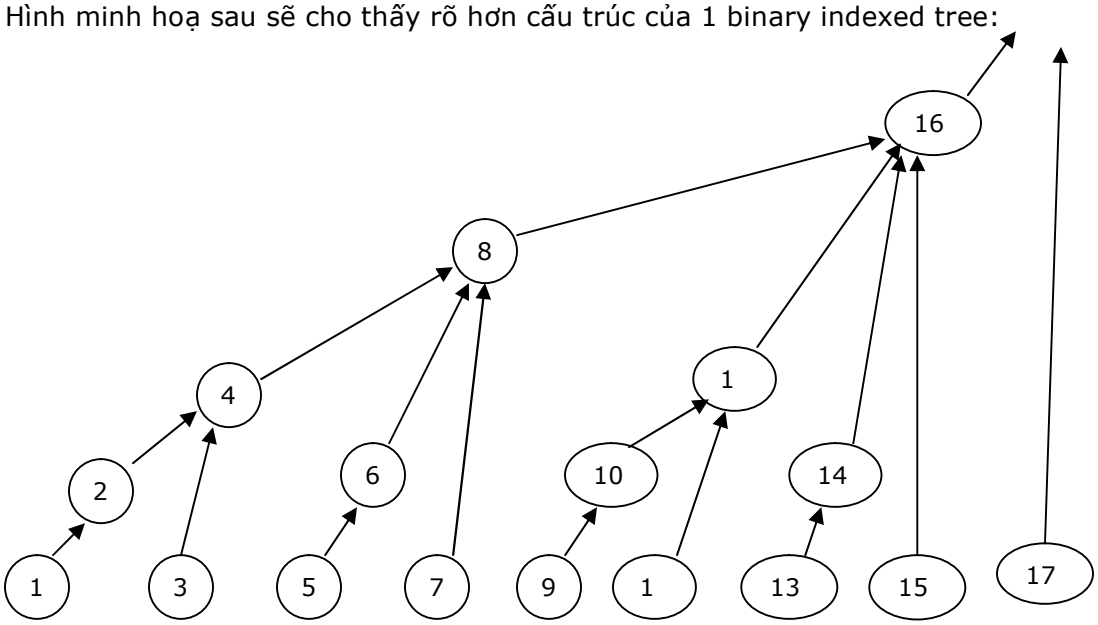
Yêu cầu: liệt kê mọi điểm trong số M điểm đã cho mà thuộc ít nhất 1 HCN.
Input:
Dòng đầu ghi 2 số nguyên N M ý nghĩa như trên.
N dòng tiếp theo mỗi dòng ghi 4 số nguyên x1 y1 x2 y2 mô tả 1 HCN với đỉnh trái dưới (x1,y1) và phải trên (x2,y2).
M dòng cuối cùng ghi toạ độ M điểm đã cho
Output: Ghi ra mọi điểm thoả mãn (thứ tự bất kì).
Giới hạn: 1<=M<=N<=20000.

4.Greatest sub sequence:
Cho dãy số A gồm N phần tử (N<=50000,|Ai|<=15000). Hàm GSS của 1 đoạn [x,y] được định nghĩa như sau: GSS(x,y)=GTLN(tổng Ai..Aj), với mọi x<=i<=j<=y. VD có dãy số {-1,2,3} thì GSS(1,2)=2, GSS(1,3)=3, GSS(2,3)=5,…
Yêu cầu: tính giá trị hàm GSS của 1 số đoạn cho trước.

II. Binary Indexed Tree.

Trong những bài toán về dãy số, 1 cấu trúc dữ liệu thường được sử dụng thay thế cho interval tree là Binary Indexed Tree. Mặc dù vậy, cấu trúc của 1 cây Binary Indexed Tree lại khác hoàn toàn với Interval Tree. Tuy gọi là "tree" nhưng có vẻ Binary Indexed Tree lại giống 1 rừng - gồm nhiều cây hơn là giống 1 cây. Cấu trúc của Binary Indexed Tree được định nghĩa 1 cách đệ quy như sau:

Binary Tree
1. i lẻ: cha[i]=i+1;
2. i chẵn: cha[i]=cha[i div 2]*2;
Như vậy cha[i] không phụ thuộc vào số nút của cây mà phụ thuộc trực tiếp vào giá trị I. Lưu ý với 1 cây N nút thì với i=1..N, cha[i]>n coi như cha[i] không tồn tại.
Hình minh hoạ sau sẽ cho thấy rõ hơn cấu trúc của 1 binary indexed tree:



Cũng như trong interval tree, thông tin được lưu ở 1 nút binary indexed tree là thông tin của nó và tất cả các nút con của nó (các phần tử ở các nút này bị nó quản lý). Thông tin ở các nút được tích luỹ dần dần lên trên. Nhưng thay vì số nút rất nhiều như ở cây interval, ta chỉ cần dùng 1 mảng O(N) để lưu trữ toàn bộ thông tin dữ liệu của cây binary indexed tree. Cải thiện về bộ nhớ này giúp ích đáng kể trong môi trường bị hạn chế bộ nhớ, đặc biệt trong Turbo Pascal khi mà bộ nhớ chỉ là 64Kb.
Thông thường sử dụng binary indexed tree chỉ sử dụng 2 lệnh cơ bản sau:
1. Từ nút I truy xuất tới nút cha[i].

2. Từ nút I truy xuất tới nút lớn nhất, nhỏ hơn I và không là con của I, gọi là I'.
Nếu căn cứ vào định nghĩa cha[i] ta có thể xác định 2 thông tin này trong O(logN) nhưng có 1 cách hiệu quả hơn nhiều:
- Để truy xuất tới cha[i] ta dựa vào công thức đã được CM sau:
  cha[i]= i + i and (i xor (i-1));
Công thức trên là công thức thường được mọi người sử dụng nhưng có những công thức hiệu quả hơn: cha[i]= i + i and (-i) = i + i and (1+not i)
- Để truy xuất tới nút đầu tiên không là con của I ta dùng công thức:
  I'= i - i and (i xor (i-1)) = i – i and (-i) = i and (i-1);
Các phép toán được dùng chỉ là +,- và các phép toán bit, thực hiện nhanh hơn nhiều so với các phép toán *,/… Đây cũng chính là điểm mạnh về tốc độ của binary indexed tree so với interval tree.
Vậy, 2 phép toán này được dùng để làm gì? Đơn giản có thể thấy việc truy xuất tới cha[i] là để cập nhật thông tin được lưu trữ. Và, ngược lại, lệnh thứ 2 chính là dùng để lấy thông tin. Dựa vào lệnh này ta có thể dễ dàng lấy được thông tin tổng hợp từ tất cả các nút từ 1 tới N: [thông tin 1..n]=A[n]+[thông tin 1..N'], trong đó A[n] là thông tin lưu trữ tại N, dấu "+" biểu hiện cho sự hợp thông tin, đôi khi có thể là phép nhân. Theo định nghĩa N' thì công thức trên đúng.
Ví dụ: [thông tin 1..15]=A[15]+A[14]+A[12]+A[8].
Độ phức tạp trong mỗi lần lấy thông tin không vượt quá O(logN+1).
Giới hạn của Binary Indexed Tree chính là vì lệnh thứ 2 chỉ tác dụng lấy thông tin trong nửa khoảng đầu tiên là 1..I mà không cho lấy thông tin tổng hợp đoan I..J bất kì trong O(logN) trong trường hợp tổng quát. Do vậy tác dụng của Binary indexed Tree cũng bị hạn chế hơn so với Interval tree. Binary Indexed Tree chỉ thực sự tuyệt vời trong các trường hợp sau:
- Thông tin được lưu trữ phải có tính tích luỹ, như tổng, tích, giá trị min, max...
- Thông tin sử dụng luôn nằm trong nửa khoảng, hoặc có tính cộng trừ nhân chia được (trong trường hợp này, thông tin có thể lấy trong các đoạn I..J bất kì trong O(logN) vì [thông tin (i..j)] = [thông tin (1..j)] – [thông tin (1..i)]).
Lưu ý:
- Binary Indexed Tree cũng có thể áp dụng đối với 1 tập hợp không cố định nhưng nếu với thông tin là tìm min/max thì độ phức tạp trong quá trình cập nhật thông tin sẽ
- Vẫn có thể lấy thông tin trong đoạn từ I..J bất kì trong các trường hợp còn lại nhưng với độ phức tạp thuật toán cao hơn 1 chút, cỡ O((logN)^2) như sau:
Dựa vào nhận xét: thông tin lưu tại I là thông tin các nút từ I'+1 tới I (theo định nghĩa I'). Do đó có thể viết 1 function tổng hợp thông tin đơn giản như sau:
Tonghop(i,j)
  1. J'=J and (J-1);
  2. If (J'+1>=I) tonghop=A[j]+tonghop(I,J')
  3. otherwise tonghop=B[j]+tonghop(I,j-1).
Trong đó B[j] là thông tin "của" J. Đôi khi cách làm này có thể thay thế cho 1 cách làm Interval tree với độ phức tạp O(logN), thời gian chạy cũng không lâu hơn là mấy nhưng code ngắn và đơn giản.
Ta cũng có thể kết hợp "Rời rạc hoá" trong sử dụng Binary Indexed Tree và mở rộng Binary Indexed Tree thành cây 2 chiều. Cách sử dụng "Rời rạc hoá" chắc đã không còn xa lạ nữa, ở đây xin nói thêm về Binary Indexed Tree 2D.
Xét bài toán xử lý trên ma trận M*N.
Chia ma trận M*N thành N dãy con, mỗi dãy con là 1 hàng. Dùng M binary indexed tree để lưu các dãy con này. Đây là tập cây đầu tiên(T1). Giá trị của 1 nút là tổng số các ô nó quản lý. Sau đó sử dụng N binary indexed tree (tập T2) để quản lý M cây trên. Cây thứ I của tập N cây T2 này sẽ quản lý tất cả các nút thứ I của M cây thuộc tập T1.

Khi đó, gặp 1 thao tác thay đổi, quá trình update lần lượt xảy ra trên tập cây T1 rồi tới tập cây T2. Giả sử update tại vị trí (U,V):

Update_BIT2D(u,v)

1. Cập nhật nút V của cây thứ U trong tập T1.
2. Cập nhật cây thứ V trong tập T2, nút bắt đầu là nút U. {qtrình này diễn ra như 1 cây bình thường}
3. t=v+v and (-v). {t=cha[v]}
4. Update_BIT2D(u,t)

END

Quá trình trên thực hiện trong O(logM*logN) vì có thủ tục gọi update cây V trong tập T2.

Còn thao tác tính giá trị HCN trái dưới (1,1) và phải trên (u,v) cũng thực hiện trong O(logM*logN) như sau:

Get2D(u,v)

1. GET+= giá trị nút U của cây thứ V, tập T2
2. v=v-v and (-v).
3. GET+=GET_BIT2D(u,v)

END

Kết quả trả về trong biến GET.

Như vậy: GET(x1,y1,x2,y2) = GET2D(x2,y2) - GET2D(x1,y2) - GET2D(x2,y1) + GET2D(x1,y1).

Trong các bài toán sử dụng cây để lưu giá trị với ý nghĩa khác ta chỉ cần sửa 1 chút trong hàm GET (bước 1) và trong hàm cập nhật cây thuộc tập T2 là được.

Vậy là ta đã biết về cách sử dụng Binary Indexed Tree 2D: hoàn toàn tương tự với cây Binary Indexed Tree bình thường, chỉ khác 1 chút trong quá trình xử lý 2 lớp cây lồng nhau.

Qua những ý trên ta đã thấy được đặc điểm của Binary Indexed Tree. Dựa vào đó cũng có thể thấy: mọi bài toán làm được bằng Binary Indexed Tree đều có thể làm được bằng Interval Tree (nhưng điều ngược lại trong 1 số trường hợp không đúng). Để hiểu rõ cấu trúc này hơn và quan trọng là ghi nhớ những công thức đã nêu, sau đây là 1 số bài tập ứng dụng. Sau khi làm bằng Binary Indexed Tree, bạn hãy thử làm với Interval Tree để so sánh tốc độ và độ phức tạp thuật toán.

Bài tập tự giải:

0. The BUS (đã được nêu trong phần bài tập về interval tree)

1. DNT – Marathon 05-06 (IOIcamp.net)

Cho dãy số A1, A2,..., An. Một nghịch thế là 1 cặp số u,v sao cho: u<v, Au>Av.

Yêu cầu: đếm số lượng nghịch thế của dãy số đã cho.

Input: Dòng đầu tiên ghi số N là số lượng số của dãy số. N dòng tiếp theo lần lượt ghi giá trị của các số thuộc dãy.

Output: 1 số duy nhất là số lượng nghịch thế đếm được.

Giới hạn:

- 1<=N<=60000
- 0<Ai<=10^9

2. MOBILE PHONES - IOI 2001.

Giả thiết một thế hệ thứ 4 điện thoại di động (mobile phone) có các trạm làm việc nằm trong vùng Tampere hoạt động như sau: Vùng hoạt động này được chia theo lưới ô vuông. Các ô vuông tạo thành một ma trận SxS với các hàng và cột được đánh số từ 0 đến S-1. Mỗi ô vuông chứa một trạm làm việc. Số lượng các điện thoại đang hoạt động (active) trong một ô vuông sẽ bị thay đổi khi người sử dụng điện thoại di chuyển từ ô này sang ô khác hoặc điện thoại chuyển chế độ bật/tắt. Theo thời gian, mỗi trạm làm việc sẽ báo cáo sự thay đổi số lượng điện thoại di động đang hoạt động trong khu vực kiểm soát của mình.

Hãy viết chương trình nhận các báo cáo đó và trả lời được các yêu cầu về tổng số điện thoại di động đang hoạt động trong một vùng không gian hình vuông cho trước.
Input:
Dòng đầu tiên ghi 0 S là kích thước bảng.
Trong 1 số dòng sau, mỗi dòng thuộc 1 trong 3 dạng sau:
- 1 X Y A : tăng thêm lượng A vào số điện thoại hoạt động trong ô vuông (x,y). A có thể là số âm
- 2 L B R T : yêu cầu cho biết tổng số lượng máy điện thoai hoạt động trong vùng HCN góc trái dưới (L,B) và phải trên (R,T).

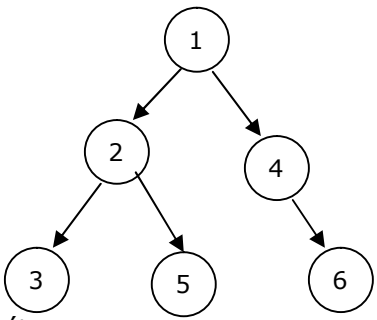
3. TEAM SELECTION – Balkan OI 2004.

Trong 1 cuộc thi lớn có N thí sinh tham gia. Cuộc thi này gồm 3 phần thi nhỏ. Tất cả N thí sinh đều tham gia và có điểm số ở cả 3 phần thi này, điểm số 2 thí sinh khác nhau trong 1 phần thi là khác nhau. Sau khi cuộc thi kết thúc, BTC muốn tìm ra các thí sinh giỏi nhất. Thí sinh giỏi nhất là thí sinh không kém hơn bất kì thí sinh nào khác. (Thí sinh A được coi là giỏi hơn thí sinh B nếu điểm số cả 3 phần thi đều cao hơn thí sinh B).
Yêu cầu: cho biết điểm 3 phần thi của N thí sinh, đếm số thí sinh được coi là giỏi nhất trong kì thi trên.
Input: dòng đầu tiên ghi số N là số thí sinh tham dự. N dòng tiếp theo dòng thứ I ghi 3 số nguyên là điểm từng môn thi của thí sinh thứ I.
Output: 1 dòng duy nhất ghi kết quả tìm được.
Giới hạn: N<=10000, điểm thi <= 10^9.

III. Heap.

Có thể nói Heap là 1 cấu trúc hữu dụng vào bậc nhất trong giải toán.
Heap là 1 cấu trúc khá quen thuộc, là 1 dạng Priority Queue (hàng đợi có độ ưu tiên), ứng dụng to lớn trong nhiều dạng toán khác nhau. Vì vậy xin chỉ nói sơ qua về Heap:
Heap thực chất là 1 cây cân bằng thoả mãn các điều kiện sau:
- 1 nút chỉ có không quá 2 nút con.
- Nút cha là nút lớn nhất, mọi nút con luôn có giá trị nhỏ hơn nút cha.

Điều kiện quan hệ nhỏ hơn của nút con so với nút cha có thể được quy định trước tuỳ theo bài toán, không nhất thiết phải là nhỏ hơn theo nghĩa toán học, ngay cả quan hệ "nút A<nút B <=> giá trị A>giá trị B" cũng hoàn toàn đúng. VD:



Mặc dù được mô tả như 1 cây nhưng Heap lại có thể lưu trữ trong mảng, nút gốc là nút 1, nút con của nút I là 2 nút 2*I và 2*I+1.
Đặc điểm của Heap:
- Nút gốc luôn là nút lớn nhất [theo định nghĩa có trước]
- Độ cao của 1 nút luôn nhỏ hơn hoặc bằng O(logN) vì cây heap cân bằng.

Ứng dụng chủ yếu của heap là chỉ tìm min, max trong 1 tập hợp động, nghĩa là tập có thể thay đổi, thêm, bớt các phần tử. (nhưng như vậy đã là quá đủ☺)
Các thao tác thường dùng trong xử lý HEAP:
-Up_heap: nếu 1 nút lớn hơn cha của nó thì di chuyển nó lên trên
-Down_heap: nếu 1 phần tử nhỏ hơn 1 con của nó thì di chuyển nó xuống dưới
-Push: đưa 1 phần tử vào HEAP bằng cách thêm 1 nút vào cây và up_heap nút đó
-Pop: loại 1 phần tử khỏi HEAP bằng cách chuyển nó xuống cuối heap và loại bỏ, sau đó chỉnh sửa lại heap sao cho thoả mãn các điều kiện của HEAP.
Sau đây là Code minh hoạ: biến top là số phần tử của heap, A là mảng chứa heap,

doicho(i,j) là thủ tục đổi chỗ 2 phần tử i và j của heap.

```
procedure Up_heap(i:longint);
begin
        if (i=1) or (a[i]>a[i div 2]) then exit; {i div 2 là nút cha của i}
        doicho(i,i div 2);{đổi chỗ 2 phần tử}
        up_heap(i div 2);
end;
procedure down_heap(i:longint);
begin
        j:=i*2;
        if j>top then exit;
        if (j<top) and (a[j]>a[j-1]) then j:=j+1; {chọn nút lớn hơn trong 2 nút con}
        doicho(i,j);
        down_heap(j);
end;
procedure push(giatri:longint);
begin
        inc(top);
        a[top]:=giatri;{mở rộng và thêm 1 phần tử vào tập}
        up_heap(top);{chỉnh lại heap cho thoả mãn điều kiện}
end;
procedure pop(vitri:longint);
begin
        a[vitri]:=a[top];
        dec(top);{loại 1 phần tử ra khỏi heap}
        {chỉnh lại heap, nếu phần tử bị loại luôn ở đầu heap có thể bỏ up_heap}
        up_heap(vitri);
        down_heap(vitri);
end;
```

1 điểm đặc biệt lưu ý là trong quá trình đưa 1 phần tử ra khỏi heap tại vị trí bất kì phải thực hiện cả 2 quá trình up_heap và down_heap để đảm bảo Heap vẫn thoả mãn điều kiện đã cho.
Qua đoạn chương trình ta có thể thấy được các điều kiện của HEAP vẫn được bảo tồn sau khi tập bị thay đổi.
Heap được sử dụng trong thuật toán Dijkstra, Kruskal, Heap Sort nhằm giảm độ phức tạp thuật toán. Heap còn có thể sử dụng trong các bài toán dãy số, QHĐ, đồ thị... Với những ví dụ sau ta sẽ thấy phần nào sự đa dạng và linh hoạt trong sử dụng Heap. Để thuận tiện ta gọi Heap-max là heap mà giá trị nút cha lớn hơn giá trị nút con (phần tử đạt max là gốc của Heap) và Heap-min là heap mà giá trị nút cha nhỏ hơn giá trị nút con (phần tử đạt min là gốc của heap).

<u>Bài toán 1: MEDIAN (phần tử trung vị).</u>

Đề bài: Phần tử trung vị của 1 tập N phần tử là phần tử có giá trị đứng thứ N div 2+1 với N lẻ và N div 2 hoặc N div 2+1 với N chẵn.
Cho 1 tập hợp ban đầu rỗng. Trong file Input có M<=10000 thao tác thuộc 2 loại:
1. PUSH gtr đưa 1 phần tử giá trị gtr vào trong HEAP (gtr<=10^9).
2. MEDIAN trả về giá trị của phần tử trung vị của tập hợp đó (nếu N chẵn trả về cả 2 giá trị).
Yêu cầu: viết chương trình đưa ra file OUTPUT tương ứng.
Input: dòng đầu tiên ghi số M, M dòng tiếp theo ghi 1 trong 2 thao tác theo định dạng trên.
Output: tương ứng với mỗi thao tác MEDIAN trả về 1 (hoặc 2) giá trị tương ứng.
Thuật giải: Dùng 2 heap, 1 heap (HA) lưu các phần tử từ thứ 1 tới N div 2 và heap

còn lại (HB) lưu các phần tử từ N div 2 +1 tới N sau khi đã sort lại tập thành tăng dần. HA là Heap-max còn HB là Heap-min. Như vậy phần tử trung vị luôn là gốc HB (N lẻ) hoặc gốc của cả HA và HB (n chẵn). Thao tác MEDIAN do đó chỉ có độ phức tạp O(1). Còn thao tác PUSH sẽ được làm trong O(logN) như sau:
- Nếu gtr đưa vào nhỏ hơn hoặc bằng HA[1] đưa vào HA ngược lại đưa vào HB. Số phần tử N của tập tăng lên 1.
- Nếu HA có lớn hơn (/nhỏ hơn N) div 2 phần tử thì POP 1 phần tử từ HA (/HB) đưa vào heap còn lại.
Sau quá trình trên thì HA và HB vẫn đảm bảo đúng theo định nghĩa ban đầu. Bài toán được giải quyết với độ phức tạp O(MlogM).

Bài toán 2: Lazy programmer – NEERC western subregion QF 2004.

Tóm tắt đề bài: Có N công việc buộc phải hoàn thành trước thời gian D[i] (thời gian hiện tại là 0). N công việc này được giao cho 1 programmer lười biếng. Xét 1 công việc I, bình thường programmer này làm xong trong B[i] thời gian nhưng nếu được trả thêm c($) thì sẽ làm xong trong B[i]-c*A[i] (nếu c=B[i]/A[i] thì anh ta có thể làm xong ngay tức khắc, t=0). Tất nhiên c<=B[i]/A[i]. Tiền trả thêm này với từng công việc là độc lập với nhau.
Yêu cầu: với các mảng D[], B[] và A[] cho trước tìm số tiền ít nhất phải trả thêm cho programmer để mọi công việc đều hoàn thành đúng hạn.
Input: Dòng đầu tiên ghi số N. Dòng thứ I trong N dòng tiếp theo mỗi dòng ghi 3 số lần lượt là A[i], B[i] và D[i].
Output: tổng số tiền nhỏ nhất phải trả thêm (chính xác tới 2 c/s thập phân).
Giới hạn: N<=10^5, 1<=A[i],B[i]<=10^4, 1<=D[i]<=10^9.
Thuật giải: Nhận thấy nếu xét tới thời điểm T thì mọi công việc có D[i]<T đều buộc phải được làm xong. Nên ta sẽ sắp xếp các công việc tăng dần theo thời gian deadline D[]. Ta chỉ phải trả thêm tiền cho programmer nếu như tới công việc thứ I tổng thời gian B[] từ 1 tới I lớn hơn D[i]. Lúc này ta cần chọn trong số các công việc trước đó 1 công việc để trả thêm tiền sao cho tiết kiệm được thời gian làm. Dĩ nhiên công việc được chọn phải có A[] càng cao càng tốt.
Từ đó ta có thuật giải sau:
1. Sắp xếp tăng dần các công việc theo các giá trị D[] của chúng
2. Dùng 1 Heap-max lưu các công việc theo giá trị A[], 1 mảng C để lưu số tiền còn có thể trả thêm cho các công việc. Khởi tạo C[i]=B[i]/A[i]. Khi xét tới công việc I thì đưa I vào Heap. Khởi tạo tien=0;
Giả sử tới công việc I thì không hoàn thành được trước D[i], cần trả thêm tiền để các công việc từ 1 tới I đều được hoàn thành đúng hạn. Ta chỉ cần trả thêm sao cho I được hoàn thành đúng D[i], giả sử đó là T. Chọn công việc đứng đầu trong heap – có A[] đạt max, giả sử là j. Lưu ý thời gian làm 1 công việc luôn dương. Có các trường hợp xảy ra là:
- C[j]*A[j]>T: C[j]-=T/A[j]; tien+=T/A[j];kết thúc xử lý công việc I.
- C[j]*A[j]=T: loại bỏ j ra khỏi heap; tien+=C[j];kết thúc;{thời gian làm j đã = 0}
- C[j]*A[j]<T: loại bỏ j ra khỏi heap; T-=C[j]*A[j]; tien+=C[j]; tiếp tục tìm công việc khác để giảm thời gian T. {thời gian làm j đã = 0}
Kết quả của bài toán chính là "tien".
Công việc trên kết thúc với T=0 nên công việc I đã được hoàn thành đúng hạn. Mọi công việc trước I đều đã hoàn thành đúng hạn nay hoặc giữ nguyên thời gian làm hoặc được trả thêm tiền làm nên cũng luôn hoàn thành đúng hạn. Vì ta luôn chọn A[] tối ưu nên số tiền phải trả cũng tối ưu. Nhờ sử dụng Heap nên độ phức tạp của thuật toán là O(NlogN) (do mỗi công việc vào và ra khỏi Heap không quá 1 lần).

Bài toán 3: Connection - 10th polish olimpiad in informatics, stage II.

Tóm tắt đề bài: Cho 1 đồ thị vô hướng gồm N đỉnh và M cung. 1 đường đi từ a tới b là đường đi đi qua các cung của đồ thị, có thể lặp lại các cung và đỉnh đã đi qua nhiều lần. Cần tìm độ dài đường đi ngắn thứ k từ a tới b cho trước.

Yêu cầu: gồm 1 số câu hỏi, mỗi câu hỏi dạng a b k phải trả về giá trị đường đi ngắn thứ k từ a tới b.
Input: Dòng đầu tiên ghi 2 số N M. Dòng thứ I trong M dòng tiếp theo mỗi dòng ghi 3 số "a b l" mô tả cung thứ I của đồ thị là cung từ a tới b có độ dài l. Dòng thứ M+2 chứa T là số câu hỏi. Trong T dòng tiếp theo mỗi dòng ghi 3 số "a b k" mô tả 1 câu hỏi. Các số trong input là số nguyên.
Output: T dòng, dòng thứ I là câu trả lời cho câu hỏi thứ I.
Giới hạn: N<=100, M<=N^2-N (đồ thị không có cung nào từ a tới a, có không quá 1 cung từ a tới b bất kì), 1<=k<=100, 0<l<=500, T<=10000. Nếu từ a tới b có nhỏ hơn k đường (đôi 1 khác nhau) thì trả về giá trị -1. VD: nếu từ 1 tới 2 có 4 đường độ dài 2,4,4 và 5 thì k=1, kết quả =2; k=2,3 kết quả =4; k=4 kết quả = 5; k>4 kết quả = -1.
Gợi ý thuật giải: Rõ ràng ta phải tính trước maxk=100 đường đi ngắn nhất từ a tới b. Làm sao để làm được điều đó? Với 1 đỉnh dùng thuật toán DIJKSTRA để tính maxk đường đi ngắn nhất tới tất cả các đỉnh còn lại. Giả sử đang xét tới đỉnh U, C[u,v,k] là đường đi ngắn thứ k từ u tới v. Với mỗi V <> U tính C[u,v,k] lần lượt với k từ 1 tới maxk (tính xong giá trị cũ rồi mới tính tới giá trị mới), k0[v] là giá trị k đang được tính của v (khởi tạo k0[v]=1). Sau đây là các bước cơ bản của thuật toán:
CONNECTION(U)
1. Với v=1..N, v<>u: Tìm v: C[u,v,k0[v]] đạt GTNN, min=C[u,v,k0[v]].
2. Xác nhận C[u,v,k0[v]] là đường cần tìm, K0[v]++.
3. Với các v' mà có đường từ v tới v' (dài L) tạo thêm 1 đường từ u tới v' độ dài L'=min+L, cập nhật đường đi từ U tới V.
End;
Các bước 1 và 3 là của thuật toán Dijkstra thông thường. Vì các giá trị min chỉ được xét 1 lần nên với mọi đường đi mới từ U tới V' ta đều phải lưu trữ lại, nhưng, do chỉ cần tìm maxk đường ngắn nhất nên ta cũng chỉ cần lưu trữ lại maxk-k0[v'] đường. bước 3 viết rõ ràng như sau:
3.Update(v',L')
3.1. Tìm đường dài nhất trong các đường đã lưu.
3.2. Nếu đường này ngắn hơn L' kết thúc.
3.3. Loại bỏ đường này.
3.4. Lưu trữ đường dài L'.
Tập các đường được lưu trữ với 1 đỉnh V là tập động, ta dùng 1 heap-max để lưu trữ tập các đường này. Lúc đó trong bước 1 thì C[u,v,k0[v]] phải chọn là min của tập trên. Có thể kết hợp 1 heap-min để tìm nhanh C[u,v,k0[v]]. Cách này cài đặt phức tạp và đòi hỏi phải hiểu rõ về heap. 1 cách khác đơn giản hơn là luôn cập nhật C[u,v,k0[v]] trong mỗi bước tìm được đường mới:
3.Update(v',L')
1.2.3.4 {các bước này như cũ}
5. Nếu (L'<C[u,v,k0[v]]) --> C[u,v,k0[v]]=L'.
Nhưng khi đó trong bước 2 của thuật toán ban đầu cần bổ sung như sau:
2.a/ Xác nhận..., K0[v]++.
b/Nếu K0[v]<maxk: Tìm C[u,v,k0[v]]=min(tập lưu trữ đường của v').
Độ phức tạp của chương trình CONNECTION là O(N*K*logK). Phải gọi N lần chương trình này nên độ phức tạp của thuật toán là O(N^2*K*logK). Lưu ý không nên dùng thuật toán Dijkstra kết hợp cấu trúc heap trong bài toán này vì đồ thị đã cho là 1 đồ thị dày.
Nhận xét: đây là 1 bài hay và khó ứng dụng heap, điểm quan trọng là nhận ra cách xây dựng lần lượt các đường ngắn nhất từ nhỏ tới lớn và ứng dụng heap vào trong quá trình này.

Qua 1 vài ví dụ trên các bạn có thể thấy phần nào ứng dụng của heap đa dạng trong

các bài toán như thế nào. Nhưng chắc không khỏi có bạn thốt lên "HEAP cũng chỉ có vậy, quá đơn giản, cứ tìm min/max thì dùng thôi". Đó là do thuật giải đã được tôi nêu rất kĩ nên đơn giản, nhưng để nghĩ được ra cách ứng dụng heap không dễ dàng như vậy. 1 số bài toán luyện tập sau sẽ giúp các bạn hiểu rõ hơn:

1. Lightest language – POI VI, stage III.

Cho trước 1 Tập Ak gồm k chữ cái đầu tiên của bảng chữ cái (2<=k<=26). Mỗi chữ cái trong tập Ak có 1 khối lượng cho trước. Khối lượng của 1 từ bằng tổng khối lượng các chữ cái trong từ đó. 1 "language" của tập Ak là 1 tập hữu hạn các từ được xây dựng chỉ bởi các chữ cái trong tập A, có khối lượng bằng tổng khối lượng các từ thuộc nó. Ta nói 1 "language" là "prefixless" nếu như với mọi cặp từ u,v trong "language" đó thì u không là tiền tố của v (u là tiền tố của v nếu tồn tại s sao cho v=u+s với '+' là phép hợp xâu).

Yêu cầu: Tìm khối lượng nhỏ nhất có thể của 1 "language" gồm đúng N từ và là 1 "prefixless" của tập Ak cho trước. (N<=10000).

Input: Dòng đầu tiên ghi 2 số N và K. Trong K dòng tiếp theo mỗi dòng ghi khối lượng của mỗi chữ cái trong tập Ak, theo thứ tự từ điển bắt đầu từ "a".

Output: Duy nhất 1 dòng ghi ra khối lượng nhỏ nhất có thể của 1 ngôn ngữ thoả những điều kiện trên.

Ví dụ:

Input

3 2

2

5

Output

16

(với input trên, ngôn ngữ được chọn là L={ab,aba,b}

2. Promotion - VII Polish Olympiad In Informatics 2000, stage III

Cho 1 tập hợp A gồm các số tự nhiên. Ban đầu tập A là tập rỗng. Trong N ngày, người ta lần lượt làm các công việc sau:

a/ Thêm vào tập A 1 số các số tự nhiên.

b/ Lưu lại hiệu giữa số lớn nhất và số nhỏ nhất của tập A.

c/ Loại bỏ 2 số lớn nhất và nhỏ nhất ra khỏi tập A.

Yêu cầu: cho biết danh sách các số được thêm vào mỗi ngày, tính tổng các số được lưu lại sau mỗi ngày. Biết trong tập A trước bước b luôn có ít nhất 2 số.

Input: Dòng đầu tiên ghi số N. Trong N dòng tiếp theo, mỗi dòng ghi theo định dạng sau: số đầu tiên là số lượng số được thêm vào, sau đó lần lượt là giá trị các số được thêm vào.

Output: 1 số duy nhất là tổng các số được lưu lại

VD:

Input:

5

3 1 2 3

2 1 1

4 10 5 5 1

0

1 2

Output:

19

Gợi ý: 1 heap-min và 1 heap-max của cùng 1 tập động, cái khó của bài toán nằm trong kĩ năng cài đặt 2 heap của cùng 1 tập. Ngoài dùng heap có thể dùng Interval Tree hoặc Binary Indexed Tree.

3. BirthDay – thi vòng 2 TH, dựa trên bài thi IOI 2005.

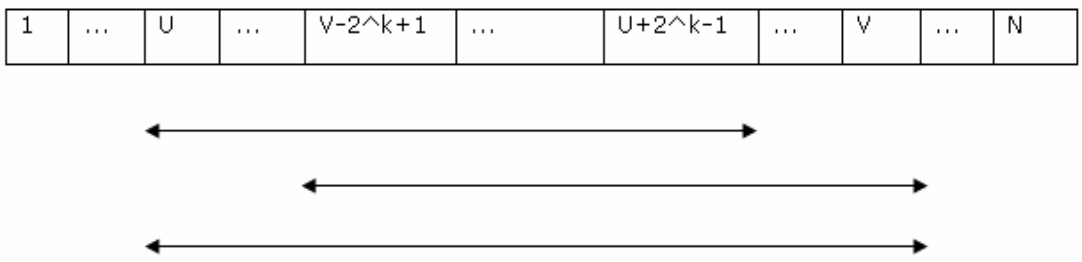
SN Byteman đã tới! Cậu đã mời được N-1 người bạn của mình tới dự tiệc SN. Cha mẹ

cậu cũng đã chuẩn bị 1 cái bàn tròn lớn dành cho N đứa trẻ. Cha mẹ của Byteman cũng biết 1 số đứa trẻ sẽ gây ồn ào, ầm ĩ nếu chúng ngồi cạnh nhau. Do đó, những đứa trẻ cần được sắp xếp lại. Bắt đầu từ Byteman, bọn trẻ được đánh số từ 1 tới N. Thứ tự mới của chúng là 1 hoán vị (p1,p2,..pn) của N số tự nhiên đầu tiên – nghĩa là sau khi xếp lại đứa trẻ p(i) ngồi giữa đứa trẻ p(i-1) và đứa trẻ p(i+1), đứa trẻ p(n) ngồi cạnh đứa trẻ p(1) và p(n-1). Để xếp lại, 1 đứa trẻ cần di chuyển 1 số bước qua trái hoặc qua phải về vị trí phù hợp. Cha mẹ của byteman muốn những đứa trẻ di chuyển càng ít càng tốt - tức là tổng độ dài di chuyển của N đứa trẻ đạt GTNN. Tìm giá trị này.
Input: Dòng đầu ghi số N, dòng tiếp theo ghi N số là thứ tự mới của bọn trẻ
Output: số bước di chuyển ít nhất thoả mãn
VD:
Input:
5
1 5 4 3 2
Output:
6

Ngoài HEAP ra còn có 1 số loại Priority Queue khác như Biominal Heap Priority Queue hay Fibonacci heap... nhưng rất phức tạp, các bạn có thể tìm hiểu thêm ở các tài liệu khác.

IV. Range Minimum Query [RMQ].

Range Minimum Query [RMQ] là 1 dạng cấu trúc đặc biệt hiệu quả trong bài toán tìm min, max của nhiều đoạn liên tiếp khác nhau của 1 dãy số cho trước.
Bài toán: Cho dãy số A gồm N số cho trước. Có 1 số câu hỏi dạng yêu cầu trả về giá trị min/max các phần tử thuộc dãy trong đoạn từ I tới J.
Không mất tổng quát ta xét bài toán tìm min.
Để giải quyết bài toán này ta có thể dùng interval tree với độ phức tạp cho mỗi yêu cầu là O(logN) và khởi tạo trong O(NlogN) nhưng 1 cách tốt hơn là dùng RMQ.
Ở đây xin đề cập tới phương pháp dùng RMQ với độ phức tạp O(NlogN) để khởi tạo và O(1) để trả lời 1 yêu cầu. Ngoài ra, cũng có 1 cách khác dùng RMQ với độ phức tạp để khởi tạo chỉ còn là O(N) nhưng khá tốn bộ nhớ và rất phức tạp.
Tư tưởng thuật toán là chia để trị. Ta cần 1 mảng RMQ với bộ nhớ O(NlogN) như sau: với mỗi i,j: R[i,j] là giá trị min trong đoạn từ i tới i+2^j-1 (1<=i<=N, 1<=j<=logN, i+2^j-1<=N). Tức là từ 1 vị trí i bất kì ta tính trước min các đoạn có độ dài là luỹ thừa của 2. Giả sử đã có mảng R, ta dễ dàng tìm giá trị min trong đoạn từ U tới V bằng cách KQ:=min(R[u,k],R[v-2^k+1,k]) trong đó k là GTLN thoả mãn 2^k<=V-U+1.



Có thể thấy đoạn [u..v] đã bị phủ bởi 2 đoạn nhỏ [u..u+2^k-1] và [v-2^k+1..v] vì tổng độ dài 2 đoạn đó là 2^k+1>v-u+1 mà 2 đoạn này chỉ gồm các phần tử thuộc trong đoạn [u..v]. Vậy giá trị KQ tìm được ở trên chính là min(u,v).
Vấn đề còn lại là khởi tạo mảng R trong O(NlogN) như thế nào? từ công thức tìm min

trên ta có thể dễ dàng suy ra cách khởi tạo mảng R: R[i,j]=min(R[i,j-1],R[i+2^j,j-1]
Đây là thủ tục khởi tạo mảng R:

```
Procedure khoitao;
Var i,j,k:longint;
Begin
        For i:=1 to N do R[i,0]:=A[i]; {với j=1}
        K:=trunc(ln(N)/ln(2));{ln(N)/ln(2)=log(N)}
        For j:=1 to k do
        For i:=1 to n-1 shl j+1 do  {1 shl j=2^j}
                If R[i,j-1]<R[i+1 shl j,j-1] then R[i,j]:=R[i,j-1]  else R[i,j]:=R[i+1 shl
j,j-1];
End;
```

Và để tìm min đoạn từ vị trí i tới vị trí j ta dùng Function FindMin sau:

```
Function Findmin(i,j:longint):longint;
Var k:longint;
Begin
        K:=trunc(ln(j-i+1)/ln(2));
        If R[i,k]<R[j-1 shl k+1,k] then FindMin:=R[i,k]
        else FindMin:=R[j-1 shl k+1,k];
End;
```

RMQ là 1 cách tiếp cận thông minh bài toán trên. Nhưng ứng dụng của RMQ không lớn và đa dạng bằng các cấu trúc khác, cách sử dụng RMQ chủ yếu là đưa bài toán trở về bài toán trên và giải bằng RMQ. Nhưng tư tưởng chia để trị như của RMQ thì có ứng dụng trong khá nhiều bài toán khác.

Sau đây là 1 ví dụ về ứng dụng của RMQ: trong bài toán tìm LCA (least common ancestor) - tổ tiên chung gần nhất của 2 nút U,V trên 1 cây cho trước. Với 2 nút U,V thì LCA(u,v) là nút cao nhất mà là tổ tiên của cả u và v.
Bài toán: cho 1 cây gồm N nút. Có 1 số yêu cầu trả về nút là tổ tiên chung gần nhất của cặp nút U,V.
Phương pháp giải:
Gọi mảng H[i] là độ cao của nút I, hay nói cách khác là độ dài đường đi từ gốc tới nút I. Đầu tiên cần chuyển bài toán tìm LCA về bài toán đã nêu. Nhận xét: trên đường đi từ U tới V thì nút có độ cao nhỏ nhất chính là cha chung của U và V.
Xét mảng B được xây dựng như sau:

```
Procedure ktB(i:longint);
Begin
        Inc(m);B[m]:=i;C[i]:=m;
        For j {thuộc tập con của i} do ktB(j);
        If cha[i]<>0 then begin Inc(m);B[m]:=cha[i];end;
End;
```

M là số phần tử của mảng B, khởi tạo m:=0;
Mảng B gọi là 1 Euler tour trên cây đã cho, xây dựng bằng cách DFS từ gốc, đi qua 1 nút thì thêm nút đó vào trong mảng B. Độ phức tạp khởi tạo mảng B bằng độ phức tạp của thuật toán DFS tức là O(N) đối với cây nếu dùng danh sách liên kết động. Số phần tử của B là m=2*n. Mảng C lưu lại vị trí của I trong mảng B. Nhận xét thấy giá trị LCA(u,v) chính là nút, hay phần tử của B trong đoạn từ C[u] tới C[v] có độ cao đạt min. Nếu như vậy dựa vào mảng B,H và dùng RMQ dễ dàng xác định được LCA(u,v). Cài đặt RMQ trong trường hợp này khác 1 chút ở điểm giá trị so sánh là H[] chứ không phải là B[], rất dễ nhầm lẫn trong cài đặt. Giả sử đã có mảng B[],C[] và H như mô tả, sau đây là code thể hiện thuật toán tìm LCA dựa vào RMQ.

```
Procedure khoitaoR;
Var i,j,k:longint;
```

```
Begin
  For i:=1 to M do R[i,0]:=B[i]; {với j=1}
  K:=trunc(ln(M)/ln(2));
  For j:=1 to k do
  For i:=1 to M-1 shl j+1 do
     If H[R[i,j-1]]<H[R[i+1 shl j,j-1]] then R[i,j]:=R[i,j-1]
     else R[i,j]:=R[i+1 shl j,j-1];
End;

Function LCA(i,j:longint):longint;
var i,j,k,tg:longint;
Begin
  I:=C[i];J:=C[j];
  If i>j then
  begin
        tg:=i;
        i:=j;
        j:=tg;
  end; {buộc i<=j}
  K:=trunc(ln(j-i+1)/ln(2));
  If H[R[i,k]]<H[R[j-1 shl k+1,k]] then LCA:=R[i,k]
  else LCA:=R[j-1 shl k+1,k];
End;
```

Sau khi khởi tạo mảng R, function LCA(i,j) sẽ trả về giá trị là LCA của 2 nút i,j.
Vấn đề còn lại là chứng minh tính đúng đắn của thuật toán trên:
Gọi T=LCA(U,V). Giả sử trong quá trình DFS, U được xét tới trước V. Dựa vào thuật toán DFS suy ra trong mảng B từ vị trí của U tới vị trí của V chắc chắn chứa T và T là đỉnh thấp nhất. Vì nếu có đỉnh thấp hơn T thì quá trình DFS đã đi ra ngoài cây gốc T nên chắc chắn đã xét qua V. Như vậy, thuật toán trên là thuật toán đúng đắn.
Độ phức tạp thuật toán là O(N) khởi tạo B + O(2Nlog(2*N)) khởi tạo RMQ và O(1) cho mỗi yêu cầu tìm LCA(u,v).
Ngoài ra, RMQ còn ứng dụng trong tìm LCP – longest common prefix của 2 suffix khi sử dụng Suffix Array xin đề cập tới trong 1 tài liệu khác nói về suffix array.
<u>Bài tập:</u>
1. Cài đặt hoàn chỉnh 2 bài toán trên.
2. Bài QTREE2 (SPOJ).
Cho trước 1 cây với các cạnh vô hướng gồm N (N<= 10000) đỉnh, các cạnh được đánh số từ 1 tới N-1. Mỗi cạnh có 1 độ dài cạnh cho trước. Viết chương trình thực hiện 1 nhóm yêu cầu thuộc 1 trong 2 kiểu sau:
DIST a b : hỏi khoảng cách giữa 2 nút a và b - tức là độ dài đường đi từ a tới b.
KTH a b k: hỏi đỉnh thứ k trên đường đi từ a tới b.
Input: Dòng đầu tiên ghi số N. Trong 1 số dòng tiếp theo mỗi dòng ghi 1 yêu cầu thuộc 1 trong 2 dạng yêu cầu trên. Input được kết thúc bởi 1 dòng ghi "DONE".
Output: Với mỗi yêu cầu ghi ra kết quả trên 1 dòng của file output.
VD:

```
Input
6
1 2 1
2 4 1
2 5 2
1 3 1
3 6 2
DIST 4 6
```

```
KTH 4 6 4
DONE
Output
5
3
```

VI. Disjoint-set.

Khi nhắc tới các cấu trúc dữ liệu trong bài toán về đồ thị ta không thể không nhắc tới Disjoint-set với nhiều ứng dụng cực kì hiệu quả.
Disjoint-set hiểu 1 cách đơn giản là 1 cách lưu trữ các tập hợp phần tử của 1 tập lớn cho trước.
Các phép toán thường được quan tâm tới trong disjoint-set là:

- MakeSet(i): tạo ra 1 tập chỉ có i.
- FindSet(i): tìm tập hợp mà nút i thuộc.
- Union(i,j): ghép 2 tập hợp chứa i và j với nhau.

Để thực hiện hiệu quả các phép toán này, disjoint-set thường được lưu trữ dưới dạng 1 rừng - tức là tập hợp của nhiều cây, mỗi cây đại diện cho 1 tập. Gốc của cây là 1 phần tử bất kì đại diện cho cả tập hợp, các phần tử khác của tập hợp được liên kết với phần tử đại diện qua 1 số liên kết. Như vậy để FindSet(i) chỉ cần tìm về gốc của cây chứa i, để Union(i,j) chỉ cần gộp 2 cây chứa i và j với nhau. Có thể lưu trữ rừng cây này bằng danh sách liên kết động hoặc mảng. Nhưng cách hay được dùng hơn và đơn giản hơn là dùng 1 mảng P[] với ý nghĩa là cha trực tiếp của nút I trong cây, với nút gốc P[] mang giá trị âm. Lệnh MakeSet(i) chỉ cần cho P[i]:=-1 là được còn lệnh FindSet và Union có thể viết đơn giản như sau:

```
Function FindSet(i:longint):longint;
Begin
  While P[i]>0 do i:=P[i];
  FindSet:=i;
End;
Procedure Union(i,j:longint);
Var u,v:longint;
Begin
  U:=FindSet(i);V:=FindSet(j);
  P[v]:=u;
End;
```

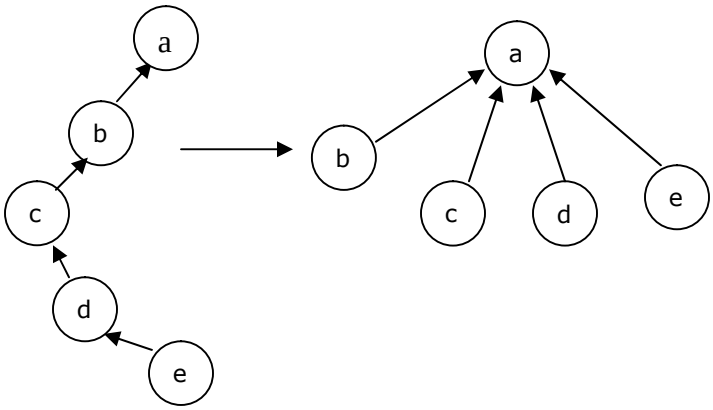
Trong trường hợp cây suy biến, hàm FindSet có thể phải thực hiện trong O(N). Vì vậy có 1 số heuristic để hỗ trợ cho disjoint-set đó là Union_by_rank và Path_compression.

- Union_by_rank thực chất là xét ưu tiên các Set khi Union chứ không Union 1 cách bất kì như trên. 1 cách ưu tiên là nối tập có số phần tử ít hơn vào tập có số phần tử nhiều hơn, như vậy hàm FindSet sẽ thực hiện nhanh hơn do độ cao của các nút trong 1 cây giảm đi. Giá trị P[] của gốc sẽ lưu số lượng nút của cây nhưng mang giá trị âm. Hàm Union được viết lại như sau:

```
Procedure Union(i,j:longint);
Var u,v:longint;
Begin
    U:=findSet(i);v:=findset(j);
    If P[u]<P[v] then
    Begin
            P[u]:=P[u]+P[v];
            P[v]:=u;
    End else
    Begin
```

```
            P[v]:=P[u]+P[v];
            P[u]:=v;
        End;
    End;
```

- Path_Compression tức là nén đường dẫn, sau thủ tục FindSet thay vì giữ nguyên các liên kết đã có, ta nối thẳng các nút trên đường đi từ I về gốc với gốc của cây. Heuristic này sẽ không sử dụng được nếu cần sử dụng quan hệ giữa i và cha của i.
- Mô tả nén đường dẫn khi FindSet từ e tới gốc a:



Hàm FindSet có thể viết lại vô cùng đơn giản như sau:

```
Function FindSet(i:longint):longint;
Begin
        If P[i]<0 then FindSet:=i else begin P[i]:=FindSet(P[i]);FindSet:=P[i];end;
End;
```

Trong bài toán cụ thể, thường chỉ cần áp dụng heuristic đầu tiên là đã đảm bảo tốc độ các lệnh của disjoint-set N phần tử cỡ O(logN), nếu dùng cả path_compression thì tốc độ sẽ vô cùng tuyệt vời, cỡ O(alpha(N))!!!. Trong đó alpha(N) là hàm Ackermann theo N, có giá trị rất nhỏ so với N, cỡ loglogN.

Như vậy ta đã biết qua về cách lưu trữ và cải tiến của Disjoint-set. Vậy ứng dụng trong giải toán của disjoint-set như thế nào? Ta sẽ xem xét 1 số bài toán cụ thể để thấy được điều này:

Bài toán 1: tìm minimum spanning tree - cây khung nhỏ nhất của 1 đồ thị cho trước. Sử dụng thuật toán Kruskal ta thấy rõ tác dụng của disjoint-set: dùng disjoint-set để lưu tập các đỉnh đã được liên kết với nhau trong thuật toán Kruskal. Vấn đề này được trình bày rất kĩ trong quyển sách "Giải thuật và Lập trình" của thầy Lê Minh Hoàng nên tôi sẽ không đề cập sâu hơn về cách thức sử dụng disjoint-set trong trường hợp này. Nếu chưa rõ các bạn có thể xem lại trong sách của thầy Hoàng.

Bài toán 2: tìm LCA - tổ tiên chung gần nhất của 2 nút trong 1 cây cho trước. Ta đã biết tới cách sử dụng RMQ để tìm LCA với độ phức tạp O(nlogn+k) với k là số yêu cầu tìm LCA. Nhưng cách sử dụng disjoint-set để tìm LCA lại là 1 cách tiếp cận hoàn toàn khác, với chi phí bộ nhớ nhỏ hơn và độ phức tạp chỉ là O(n*alpha(N)+k) nếu sử dụng 2 heuristic đã nêu. Cách này có phức tạp hơn 1 chút trong cài đặt. Thuật toán tìm LCA dựa vào disjoint-set được gọi là thuật toán Tarjan.

Nếu như sử dụng RMQ tìm LCA với mỗi yêu cầu ta trả về 1 giá trị LCA, hay gọi cách khác là xử lý online các yêu cầu thì trong thuật toán Tarjan, các yêu cầu lại được xử lý theo lối offline. Xử lý offline nghĩa là toàn bộ các yêu cầu sẽ được đọc vào và lưu lại, quá trình xử lý sau sẽ song song tìm lời giải cho các yêu cầu, mà ở đây là tìm LCA 2 nút i,j, tất cả các kết quả được lưu lại và trả lời theo thứ tự yêu cầu. Lý do là thuật toán không thể đảm bảo các yêu cầu được trả lời đúng theo thứ tự nên cần lưu lại và xử lý offline.

Vậy thuật toán Tarjan xử lý offline các yêu cầu như thế nào? Đầu tiên, các yêu cầu được lưu lại dưới dạng danh sách yêu cầu: mỗi yêu cầu tìm LCA(i,j) thì trong danh sách yêu cầu DS[i] của i thêm 1 nút j và trong danh sách yêu cầu DS[j] của j lưu 1 nút i.
Thuật toán Tarjan duyệt cây theo thứ tự duyệt DFS, sau khi xét xong 1 nút sẽ xử lý danh sách yêu cầu của nút đó. Bằng cách nào để xử lý được danh sách yêu cầu? Câu trả lời chính là dùng Disjoint-set:
Thuật toán Tarjan có thể được mô tả bởi đoạn code giả sau:
Mảng bool DX[] cho biết nút I đã được xét xong hay chưa, khởi tạo là False
Mảng int Ancestor[] với gốc 1 tập cho biết nút thấp nhất trong tập đó

```
{
LCA(i)
    1   makeSet(i)
    2   Ancestor[i]=i
    3   For j {là con của i} do
            4   LCA(j)
            5   Union(i,j)
            6   Ancestor[FindSet(i)]=i
    7   Dx[i]=true
    8   For j {thuộc DS[i]} do
                If Dx[j] then LCA(i,j)=Ancestor[FindSet(j)]
}
```

CM tính đúng đắn của thuật toán:
Đầu tiên ta nhận thấy mỗi cặp LCA(i,j) chỉ được xử lý 1 lần, do trong i và j luôn có 1 nút được xét xong trước và không được xét lại nữa. Và, 1 tập hợp luôn là 1 cây mà gốc là Ancestor[] của tập đó theo câu lệnh 6.
Thứ 2 là số tập Set đơn lẻ khi xét tới nút I chính bằng độ cao của i trong cây, hơn thế nữa mảng Ancestor của các tập này lập thành đường đi từ gốc tới I. Thực vậy, từ câu lệnh 5 ta thấy khi đã xét tới i thì mọi nút là tổ tiên của I đều chưa được xét xong, nên có ít nhất H[i] tập. Ngoài ra, mọi nút không là tổ tiên của I mà đã được xét thì đều đã xét xong, được Union vào tập chứa cha trực tiếp của nó, không làm tăng thêm số tập. Như vậy nhận xét vừa nêu là hoàn toàn chính xác.
Ta quan tâm tới tính đúng đắn của câu lệnh số 8. Với j đã xét xong, có 2 trường hợp xảy ra:
a/ J thuộc cây con gốc I --> FindSet(j)=i, Ancestor[i]=i --> LCA(i,j)=i. kết quả đúng
b/ J không thuộc cây con gốc I. LCA(i,j) phải là 1 tổ tiên của I --> nếu k=Ancestor[FindSet(j)] thì k là 1 tổ tiên của I theo nhận xét trên. K cũng là tổ tiên của j. --> K là tổ tiên chung của I và J. Giả sử LCA(i,j)=K'<>K suy ra: K là tổ tiên của K', J đã được xét xong nên J và K' chắc chắn thuộc cùng 1 tập, suy ra K và K' thuộc cùng 1 tập (*). Mà K' vẫn chưa được xét xong, vì là tổ tiên của I, mâu thuẫn với giả thiết (*). Do đó K=LCA(i,j).
Thuật toán đã được chứng minh.
Độ phức tạp thuật toán là O(N*alpha(N)) để xây dựng tập disjoint-set và O(2K) để trả lời k yêu cầu. Như vậy độ phức tạp thuật toán là O(N*alpha(N)+k).

Bài toán 3: Bài THE GANGS – BOI 2003
Trong những năm 1920, Chicago là chiến trường của những tay Gangster. 2 Gangster gặp nhau thì hoặc sẽ là bạn hoặc là thù. Trong giới Gangster chỉ có 2 luật lệ:
1. Bạn của bạn là bạn.
2. Thù của thù là bạn.
2 Gangster khác nhau thuộc cùng 1 nhóm khi và chỉ khi chúng là bạn của nhau.
Cho biết trước M số quan hệ bạn và thù của N gangster. Tìm số nhóm gangster

nhiều nhất thoả mãn các điều kiện trên.
Input: Dòng đầu tiên ghi số nguyên N, dòng tiếp theo ghi số nguyên M là số quan hệ. Trong M dòng tiếp theo mỗi dòng thuộc 1 trong 2 dạng: E u v nghĩa là u,v là kẻ thù hoặc F u v nghĩa là u,v là bạn.
Output: số nhóm nhiều nhất có thể.
VD:
```
Input
6
4
E 1 4
F 5 3
F 4 6
E 1 2
Output
3
```
Gợi ý: Có vẻ thuật toán quá đơn giản! Chỉ cần xem xét 2 Gangster nếu là bạn thì gộp 2 nhóm chứa chúng thành 1!! Đúng là vậy nhưng nếu làm 1 cách thô thiển độ phức tạp thuật toán sẽ lên tới O(N) cho mỗi lần gộp 2 nhóm gangster. Cần phải chú ý rằng không có quan hệ "bạn của thù là thù" vì vậy không thể sử dụng 1 thuật toán tìm TPLT đơn thuần. Ta tìm cách giảm độ phức tạp của quá trình gộp 2 nhóm Gangster, và Disjoint-set là lựa chọn. Đơn giản chỉ cần khởi tạo mỗi gangster là 1 nhóm, nếu chúng là bạn, hoặc thù của thù thì gộp 2 nhóm lại bằng thủ tục Union. Để ý rằng tất cả kẻ thù của 1 gangster đều thuộc 1 nhóm vì vậy chỉ cần dùng 1 mảng thu[] ý nghĩa thù[i] với lưu 1 kẻ thù của gangster thuộc vào. Như vậy, với mỗi quan hệ:
1. I J là bạn --> Union(Findset(i),Findset(j));
2. I J là thù --> if (thu[i]=0) thu[i]=j otherwise Union(findset(i),findset(thu[j])).
Tương tự với J.
Độ phức tạp thuật toán chỉ còn là O(M*alpha(N)).
Phần cài đặt thuật toán chắc không còn gì khó khăn nữa.

<u>Bài toán luyện tập:</u>
1. Giải lại bài QTREE2 trong phần RMQ bằng disjoint-set.
2. Parity:
Cho một dãy nhị phân có n phần tử. (n<=1.000.000.000) và m câu phát biểu (m<=5000)
Mỗi câu phát biểu có dạng a b c. Có ý nghĩa là đoạn từ a đến b có tổng các số 1 là lẻ nếu c=1 và chẵn nếu c=0.
Hãy xác định xem các phát biểu ấy còn đúng đến câu số mấy (tức là tồn tại ít nhất 1 dãy nhị phân n phần tử thoả mãn tất cả các điều kiện tới câu đó).
Input: Dòng đầu ghi số N, dòng tiếp theo ghi số M. Trong M dòng tiếp theo ghi 3 số a b c mô tả 1 phát biểu.
Output: Duy nhất 1 số là số lớn nhất mà tới câu phát biểu đó thì mọi phát biểu từ đó trở lên đều đúng.
VD:
```
Input
10
5
1 2 0
3 4 1
5 6 0
1 6 0
7 10 1
```

Output
3

Tài liệu tham khảo:

- Introdution to Algorithms 2nd Edition.
- The LCA problem revisited, Michael A.Bender and Martin Farach-Colton.
- Giải thuật và lập trình, Lê Minh Hoàng.