## Câu 1: Quick Sort

Ý Tưởng

Quick sort là một trong những thuật toán chia để trị. Quick sort chia một mảng lớn của chúng ta thành hai mảng con nhỏ hơn: mảng có phần tử nhỏ và mảng có phần tử lớn. Sau đó Quick sort có thể sort các mảng con này bằng phương pháp đệ quy. Các bước ý tưởng của Quick sort là:

1. Chọn một phần tử để so sánh, tôi gọi đây là phần tử Key hoặc Pivot tùy vào mỗi người, từ trong mảng đầu tiên của chúng ta.
2. Sau đó phân vùng và sort mảng con của sau phân vùng của chúng ta làm sao cho các phần tử lớn hơn phần tử Key nằm sau(bên phải) và các phần tử bé hơn phần tử Key nằm trước(bên trái). Đây được gọi là quá trình phân vùng.
3. Cuối cùng là đệ quy sử dụng các bước trên cho các mảng với phần tử bé hơn và phân tách với các phần tử lớn hơn sau khi phân vùng.

Các cách chọn phần tử Key

1. Chọn phần tử bên trái ngoài cùng (phần tử đầu tiên của mảng).
2. Chọn phần tử bên phải ngoài cùng (phần tử cuối cùng của mảng).
3. Ngẫu nhiên một phần tử (sử dụng hàm random mà ngôn ngữ lập trình cung cấp).
4. Phần tử chính giữa của mảng (chiều dài của mảng chia đôi).

V. **Quick sort**  
Đây là giả thuật sắp xếp nhanh, chỉ tốn O(nlogn). Cài đặt của giả thuật tương đối phức tạp. Chúng ta cần chú ý đến:

1. Pivot: ta gọi là chốt
2. Partition: Gọi là điểm phân hoạch

Đối với giải thuật này, chúng ta xem 1 mảng 1 phần tử hay mảng có tất cả phần tử giống nhau là một mảng có thứ tự. Ta sẽ chi mảng thành 2 mảng con: Trái và phải. Sắp xếp 2 mảng con, và cứ như thế cho đến khi mảng còn 1 phần tử. Nếu mảng Trái đã có thứ tự và mảng phải cũng có thứ tự thì mảng của chúng ta đã là mảng có thứ tự.  
  
A = {59, 31, 12, 33, 27, 97, 91, 19, 18, 63 }  
  
Bước 1: Tìm chốt: Chốt là phần tử lớn nhất trong 2 phần tử khác nhau đầu tiên của mảng. Nếu mảng có 1 phần tử tất cả phần tử bằng nhau sẽ không có chốt:  
Chốt của mảng trên la 59 (vị trí 0).  
VD:   
+ 1, 1, 5, 3, 2 -> chốt sẽ là 5  
+ 5, 5, 5, 3, 1, 2, 3 -> chốt là 5  
+ 6 -> không có chốt  
+ 7, 7, 7, 7 -> không có chốt  
  
Bước 2: Tìm điểm phân hoạch:   
+ Dùng 2 cờ: L (trái) và R (Phải)  
+ L chạy từ trái qua, dừng lại khi gặp phần tử >= pivot  
+ R chạy từ phải qua, dừng lại khi gặp phần tử < pivot  
+ Tại điểm dùng: Nếu L < R : Chúng ta sẽ swap A[L] và A[R]  
+ Dừng lại khi L > R  
+ Partition sẽ là L. Đây sẽ là chỉ số đầu tiên của mảng bên phải  
VD: với mảng A = {59, 31, 12, 33, 27, 97, 91, 19, 18, 63 }  
PivotLey = 59  
L = 0, R = 9:  
b1. L dừng lại tại vị trí 0: vì A[L] >= pivot, R dừng lại tại vị trí 8, vì A[R] = 18 < pivot.  
Swap: **18**, 31, 12, 33, 27, 97, 91, 19, **59**, 63   
b2, tiếp tục cho đến khi L > R  
Bước 3: Lặp lại như thế (Dùng đệ qui)

Độ phức tạp:

– Trường hợp tốt nhất: O(nlog2n)  
– Trường hợp xấu nhất: O(n2)  
– Trường hợp trung bình: O(nlog2n)

## Câu 2: MergeSort

Ý tưởng chúng ta sẽ chia mảng lớn thành những mảng con nhỏ hơn bằng cách chia đôi mảng lớn và chúng ta tiếp tục chia đôi các mảng con cho tới khi mảng con nhỏ nhất chỉ còn 1 phần tử. Sau đó chúng ta sẽ tiếng hành so sánh 2 mảng con có cùng mảng cơ sở (khi chúng ta chia đôi mảng lớn thành 2 mảng con thì mảng lớn đó chúng ta gọi là mảng cơ sở của 2 mảng con đó) khi so sánh chúng sẽ vừa sắp xếp vừa ghép 2 mảng con đó lại thành mảng cơ sở, chúng ta tiếp tục so sánh và ghép các mảng con lại đến khi còn lại mảng duy nhất thì đó là mảng đã được sắp xếp.

* Hiệu quả được tính bằng công thức O(n log n).

Một cách gọi [đệ quy](https://vi.wikipedia.org/wiki/%C4%90%E1%BB%87_quy) của sắp xếp trộn cũng thường được hướng dẫn trong các giáo trình giải thuật.

Để sắp xếp trộn đoạn {\displaystyle a[k\_{1}..k\_{2}]} của danh sách {\displaystyle a[1..n]} ta chia đoạn đó thành 2 phần {\displaystyle a[k\_{1}..k\_{3}]} và {\displaystyle a[k\_{3}+1..k\_{2}]},trong đó {\displaystyle k\_{3}=int((k\_{1}+k\_{2})/2)} tiến hành sắp xếp với mỗi phần rồi trộn chúng lại. Lời gọi thủ tục sắp xếp trộn với {\displaystyle a[1..n]} sẽ cho kết quả là sắp toàn bộ danh sách {\displaystyle a[1..n]}

Ví dụ: Cho danh sách {\displaystyle a=[2,7,6,3,4,5,1]}

Giải thuật trộn đệ quy chia a thành hai danh sách con và tiến hành 3 bước

|  |  |
| --- | --- |
| Danh sách trái | Danh sách phải |
| 2,7,6 | 3,4,5,1 |

* Sắp xếp trộn danh sách trái 2,7,6

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Quá trình chia | | |  | Quá trình trộn | | |
| 2,7,6 | | |  | 2,6,7 | | |
| 2 | 7,6 | |  | 2 | 6,7 | |
| 2 | 7 | 6 |  | 2 | 7 | 6 |

* Sắp xếp trộn danh sách phải 3,4,5,1

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Quá trình chia | | | |  | Quá trình trộn | | | |
| 3,4,5,1 | | | |  | 1,3,4,5 | | | |
| 3,4 | | 5,1 | |  | 3,4 | | 1,5 | |
| 3 | 4 | 5 | 1 |  | 3 | 4 | 5 | 1 |

* Trộn danh sách trái 2,6,7 với danh sách phải 1,3,4,5

|  |  |  |
| --- | --- | --- |
| Danh sách trái | Danh sách phải | Danh sách trộn |
| 2,6,7 | 1,3,4,5 | 1,2,3,4,5,6,7 |

## Câu 3: Giải Thuật Radix Sort

|  |
| --- |
|   **Ý tưởng thuật toán**  o   Khác với các thuật toán trước, Radix sort là một thuật toán tiếp cận theo một hướng hoàn toàn khác. Nếu như trong các thuật toán khác, cơ sở để sắp xếp luôn là việc so sánh giá trị của 2 phần tử thì Radix sort lại dựa trên nguyên tắc phân loại thư của bưu điện. Vì lý do đó nó còn có tên là Postmans sort. Nó không hề quan tâm đến việc so sánh giá trị của phần tử và bản thân việc phân loại và trình tự phân loại sẽ tạo ra thứ tự cho các phần tử.  o   Ta biết rằng, để chuyển một khối lượng thư lớn đến tay người nhận ở nhiều địa phương khác nhau, bưư điện thường tổ chức một hệ thống phân loại thư phân cấp. Trước tiên, các thư đến cùng một tỉnh, thành phố sẽ được sắp chung vào một lô để gửi đến tỉnh thành tương ứng. Bưu điện các tỉnh thành này lại thực hiện công việc tương tự. Các thư đến cùng một quận, huyện sẽ được xếp vào chung một lô và gửi đến quận, huyện tương ứng. Cứ như vậy, các bức thư sẽ được trao đến tay người nhận một cách có hệ thông mà công việc sằp xếp thư không quá nặng nhọc.  o   Mô phỏng lại qui trình trên, để sắp xếp dãy a1, a2, …, an, giải thuật Radix Sort thực hiện như sau:    Trước tiên, ta có thể giả sử mỗi phần tử ai trong dãy a1, a2, …, an là một số nguyên có tối đa m chữ số.    Ta phân loại các phần tử lần lượt theo các chữ số hàng đơn vị, hàng chục, hàng trăm, . tương tự việc phân loại thư theo tỉnh thành, quận huyện, phường xã, ..  o   Các bước thực hiện thuật toán như sau:    Bước 1 :  // k cho biết chữ số dùng để phân loại hiện hành           k = 0;    // k = 0: hàng đơn vị; k = 1:hàng chục;    Bước 2 : //Tạo các lô chứa các loại phần tử khác nhau           Khởi tạo 10 lô B0, B1, ., B9 rỗng;    Bước 3 :           For i = 1 .. n do           Ðặt ai vào lô Bt với t = chữ số thứ k của ai;    Bước 4 :           Nối B0, B1, ., B9 lại (theo đúng trình tự) thành a.    Bước 5 :           k = k+1;           Nếu k < m thì trở lại bước 2.           Ngược lại: Dừng |

 **Độ phức tạp**

* Với một dãy n số, mỗi số có tối đa m chữ số, thuật toán thực hiện m lần các thao tác phân lô và ghép lô.
* Trong thao tác phân lô, mỗi phần tử chỉ được xét đúng một lần, khi ghép cũng vậy.
* Như vậy, chi phí cho việc thực hiện thuật toán hiển nhiên là O(2mn) = O(n).
* Cho dãy số a:
* 701 1725 999 9170 3252 4518 7009 1424 428 1239 8425 7013
* **Phân lô theo hàng đơn vị:**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 12 | 0701 |  |  |  |  |  |  |  |  |  |  |
| 11 | 1725 |  |  |  |  |  |  |  |  |  |  |
| 10 | 0999 |  |  |  |  |  |  |  |  |  |  |
| 9 | 9170 |  |  |  |  |  |  |  |  |  |  |
| 8 | 3252 |  |  |  |  |  |  |  |  |  |  |
| 7 | 4518 |  |  |  |  |  |  |  |  |  |  |
| 6 | 7009 |  |  |  |  |  |  |  |  |  |  |
| 5 | 1424 |  |  |  |  |  |  |  |  |  |  |
| 4 | 0428 |  |  |  |  |  |  |  |  |  |  |
| 3 | 1239 |  |  |  |  |  |  |  |  |  | 0999 |
| 2 | 8425 |  |  |  |  |  | 1725 |  |  | 4518 | 7009 |
| 1 | 7013 | 9170 | 0701 | 3252 | 7013 | 1424 | 8425 |  |  | 0428 | 1239 |
| CS | B | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

* **Phân lô theo hàng chục:**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 12 | 0999 |  |  |  |  |  |  |  |  |  |  |
| 11 | 7009 |  |  |  |  |  |  |  |  |  |  |
| 10 | 1239 |  |  |  |  |  |  |  |  |  |  |
| 9 | 4518 |  |  |  |  |  |  |  |  |  |  |
| 8 | 0428 |  |  |  |  |  |  |  |  |  |  |
| 7 | 1725 |  |  |  |  |  |  |  |  |  |  |
| 6 | 8425 |  |  |  |  |  |  |  |  |  |  |
| 5 | 1424 |  |  |  |  |  |  |  |  |  |  |
| 4 | 7013 |  |  | 0428 |  |  |  |  |  |  |  |
| 3 | 3252 |  |  | 1725 |  |  |  |  |  |  |  |
| 2 | 0701 | 7009 | 4518 | 8425 |  |  |  |  |  |  |  |
| 1 | 9170 | 0701 | 7013 | 1424 | 1239 |  | 3252 |  | 9170 |  | 0999 |
| CS | A | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

* **Phân lô theo hàng trăm:**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 12 | 0999 |  |  |  |  |  |  |  |  |  |  |
| 11 | 9170 |  |  |  |  |  |  |  |  |  |  |
| 10 | 3252 |  |  |  |  |  |  |  |  |  |  |
| 9 | 1239 |  |  |  |  |  |  |  |  |  |  |
| 8 | 0428 |  |  |  |  |  |  |  |  |  |  |
| 7 | 1725 |  |  |  |  |  |  |  |  |  |  |
| 6 | 8425 |  |  |  |  |  |  |  |  |  |  |
| 5 | 1424 |  |  |  |  |  |  |  |  |  |  |
| 4 | 4518 |  |  |  |  |  |  |  |  |  |  |
| 3 |  |  |  |  |  | 0428 |  |  |  |  |  |
| 2 | 7009 | 7013 |  | 3252 |  | 8425 |  |  | 1725 |  |  |
| 1 | 0701 | 7009 | 9170 | 1239 |  | 1424 | 4518 |  | 0701 |  | 0999 |
| CS | B | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

* **Phân lô theo hàng ngàn:**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 12 | 0999 |  |  |  |  |  |  |  |  |  |  |
| 11 | 1725 |  |  |  |  |  |  |  |  |  |  |
| 10 | 0701 |  |  |  |  |  |  |  |  |  |  |
| 9 | 4518 |  |  |  |  |  |  |  |  |  |  |
| 8 | 0428 |  |  |  |  |  |  |  |  |  |  |
| 7 | 8425 |  |  |  |  |  |  |  |  |  |  |
| 6 | 1424 |  |  |  |  |  |  |  |  |  |  |
| 5 | 3252 |  |  |  |  |  |  |  |  |  |  |
| 4 | 1239 |  |  |  |  |  |  |  |  |  |  |
| 3 | 9170 | 0999 | 1725 |  |  |  |  |  |  |  |  |
| 2 | 7013 | 0701 | 1424 |  |  |  |  |  | 7013 |  |  |
| 1 | 7009 | 0428 | 1239 |  | 3252 | 4518 |  |  | 7009 | 8425 | 9170 |
| CS | B | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

* **Lấy các phần tử từ các lô B0, B1, ., B9 nối lại thành a:**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 12 | 9170 |  |  |  |  |  |  |  |  |  |  |
| 11 | 8425 |  |  |  |  |  |  |  |  |  |  |
| 10 | 7013 |  |  |  |  |  |  |  |  |  |  |
| 9 | 7009 |  |  |  |  |  |  |  |  |  |  |
| 8 | 4518 |  |  |  |  |  |  |  |  |  |  |
| 7 | 3252 |  |  |  |  |  |  |  |  |  |  |
| 6 | 1725 |  |  |  |  |  |  |  |  |  |  |
| 5 | 1424 |  |  |  |  |  |  |  |  |  |  |
| 4 | 1239 |  |  |  |  |  |  |  |  |  |  |
| 3 | 0999 |  |  |  |  |  |  |  |  |  |  |
| 2 | 0701 |  |  |  |  |  |  |  |  |  |  |
| 1 | 0428 |  |  |  |  |  |  |  |  |  |  |
| CS | B | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

## Câu 4: Bucket Sort

**  ***Ý tưởng thuật toán***

*+ Xét dãy A có n khóa và miền giá trị của các phần tử là [0, m-1].  
+ Sử dụng một mảng B gọi là mảng thẻ (bucket array). Mảng thẻ có m phần tử.  
+ Sử dụng giá trị của A chính là chỉ số trong mảng B.  
+ Đặt các phần tử của A vào B với vị trí tương ứng với giá trị của nó.*

**  ***Độ phức tạp***

o   Vòng for đầu tiên: O(n)

o   Vòng for thứ hai: O(m)

 O(m+n)

Nếu m tỉ lệ với n: m = cn thì độ phức tạp là O(n + cn)= O(n)  độ phức tạp là tuyến tính

## Câu 5: Select\_k

## [Thuật toán sắp xếp chọn (Selection Sort)](http://cnttk25.forumvi.com/t6-topic#8)

Bài gửi by [hosytan](http://cnttk25.forumvi.com/u2) on 9/5/2013, 11:13

**1. Bài toán**  
Cho dãy X = {X1, X1, ..., Xn}, hãy sắp xếp dãy theo chiều không giảm.  
  
**2. Ý tưởng**  
Chọn phần tử nhỏ nhất trong n phần tử ban đầu, đưa phần tử này về vị trí đúng là đầu tiên của dãy hiện hành. Sau đó không quan tâm đến nó nữa, xem dãy hiện hành chỉ còn n-1 phần tử của dãy ban đầu, bắt đầu từ vị trí thứ 2. Lặp lại quá trình trên cho dãy hiện hành đến khi dãy hiện hành chỉ còn 1 phần tử.   
Do dãy ban đầu có n phần tử, vậy tóm tắt ý tưởng thuật toán là thực hiện n-1 lượt việc đưa phần tử nhỏ nhất trong dãy hiện hành về vị trí đúng ở đầu dãy.  
  
**3. Giải thuật**  
- Bước 1: i = 1;  
- Bước 2: Tìm X[min] trong dãy X[i] ... X[n]  
- Bước 3: Đỗi chổ X[i] cho X[min], nếu min trùng với i thì bỏ qua bước này.  
- Bước 4:   
\* Nếu i <= n-1 thì tăng i lên 1 vị trí (i = i +1), quay lại Bước 2.  
\* Ngược lại, dừng, dãy đã cho đã sắp xếp đung vị trí.

**Bước 1**: Thiết lập MIN về vị trí 0

**Bước 2**: Tìm kiếm phần tử nhỏ nhất trong danh sách

**Bước 3**: Tráo đổi với giá trị tại vị trí MIN

**Bước 4**: Tăng MIN để trỏ tới phần tử tiếp theo

**Bước 5**: Lặp lại cho tới khi toàn bộ danh sách đã được sắp xếp

## Cách giải thuật sắp xếp chọn (Selection Sort) làm việc

Dưới đây là các hình minh họa cho cách giải thuật sắp xếp chọn làm việc. Giả sử chúng ta có một mảng như sau:

Giải thuật sắp xếp chọn (Selection Sort)

Từ vị trí đầu tiên trong danh sách đã được sắp xếp, toàn bộ danh sách được duyệt một cách liên tục. Vị trí đầu tiên có giá trị 14, chúng ta tìm toàn bộ danh sách và thấy rằng 10 là giá trị nhỏ nhất.

Giải thuật sắp xếp chọn (Selection Sort)

Do đó, chúng ta thay thế 14 với 10. Sau một vòng lặp, giá trị 10 thay thế cho giá trị 14 tại vị trí đầu tiên trong danh sách đã được sắp xếp. Chúng ta tráo đổi hai giá trị này.

Giải thuật sắp xếp chọn (Selection Sort)

Tại vị trí thứ hai, giá trị 33, chúng ta tiếp tục quét phần còn lại của danh sách theo thứ tự từng phần tử.

Giải thuật sắp xếp chọn (Selection Sort)

Chúng ta thấy rằng 14 là giá trị nhỏ nhất thứ hai trong danh sách và nó nên xuất hiện ở vị trí thứ hai. Chúng ta tráo đổi hai giá trị này.

Giải thuật sắp xếp chọn (Selection Sort)

Sau hai vòng lặp, hai giá trị nhỏ nhất đã được đặt tại phần đầu của danh sách đã được sắp xếp.

Giải thuật sắp xếp chọn (Selection Sort)

Tiến trình tương tự sẽ được áp dụng cho phần còn lại của danh sách. Các hình dưới minh họa cho các tiến trình này.



Tiếp theo chúng ta sẽ theo dõi một số khía cạnh khác của giải thuật sắp xếp chọn.

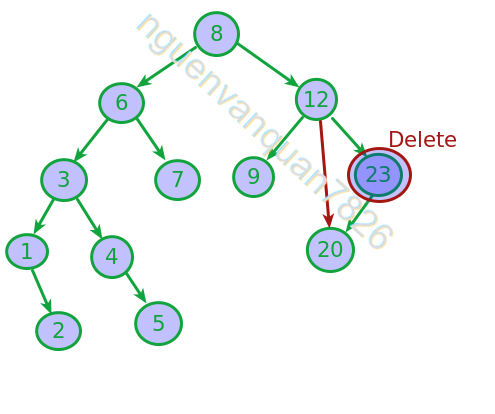
huật toán ít phải đổi chỗ các phần tử nhất trong số các thuật toán sắp xếp(n lần hoán vị) nhưng có độ phức tạp so sánh là O(n2) (n2/2 phép so sánh).

## Câu 6: Thêm 1 phần tử vào cây

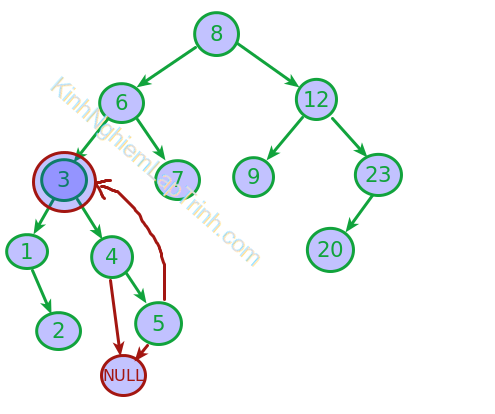
Việc thêm một phần tử X vào cây phải bảo đảm điều kiện ràng buộc của CNPTK. Ta có thể thêm vào nhiều chỗ khác nhau trên cây, nhưng nếu thêm vào một nút lá sẽ là tiện lợi nhất do ta có thể thực hiên quá trình tương tự thao tác tìm kiếm. Khi chấm dứt quá trình tìm kiếm cũng chính là lúc tìm được chỗ cần thêm.

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14 | int insertNode(Tree &T, item x) // chen 1 Node vao cay  {      if (T != NULL)      {          if (T->key == x) return -1;  // Node nay da co          if (T->key > x) return insertNode(T->Left, x); // chen vao Node trai          else if (T->key < x) return insertNode(T->Right, x);   // chen vao Node phai      }      T = (Node \*) malloc(sizeof(Node));      if (T == NULL) return 0;    // khong du bo nho      T->key = x;      T->Left = T->Right = NULL;      return 1;   // ok  } |

Vậy là xong con ong.

Việc hủy một phần tử X ra khỏi cây phải bảo đảm điều kiện ràng buộc của CNPTK. Có 3 trường hợp khi hủy nút X có thể xảy ra:  
– X là nút lá: Đơn giản chỉ cần hủy X vì nó không móc nối đến phần tử nào khác.  


– X chỉ có 1 con (trái hoặc phải): Trước khi hủy X ta móc nối cha của X với con duy nhất của nó.  
– X có đủ cả 2 con: ta không thể hủy trực tiếp do X có đủ 2 con. Ta sẽ hủy gián tiếp. Thay vì hủy X, ta sẽ tìm một phần tử thế mạng Q. Phần tử này có tối đa một con. Thông tin lưu tại Q sẽ được chuyển lên lưu tại X. Sau đó, nút bị hủy thật sự sẽ là Y giống như 2 trường hợp đầu. Vấn đề là phải chọn Y sao cho khi lưu Q vào vị trí của X, cây vẫn là CNPTK.  
Có 2 phần tử thỏa mãn yêu cầu:  
+ Phần tử nhỏ nhất (trái nhất) trên cây con phải.  
+ Phần tử lớn nhất (phải nhất) trên cây con trái.  
Việc chọn lựa phần tử nào là phần tử thế mạng hoàn toàn phụ thuộc vào ý thích của người lập trình. Trong code này tôi chọn phần tử phải nhất.



|  |  |
| --- | --- |
|  | độ phức tạp của thuật toán này được xem là O(log2(N)) |

**Câu 7: cây đỏ đen (Red-Black tree)**

# Các phép toán trên cây đỏ đen

Có thể áp dụng ngay các phép chèn, xóa trong cây tìm kiếm nhị phân vào cây đỏ đen mà không cần sửa chữa gì vì cây đỏ đen là trường hợp riêng của cây tìm kiếm nhị phân. Tuy nhiên, khi đó có thể có một số tính chất trong định nghĩa của cây đỏ đen sẽ bị vi phạm. Việc khôi phục các tính chất đỏ đen sẽ cần một số nhỏ cỡ O(log n) hoặc trung bình chỉ O(1) các phép đổi màu (tốn rất ít thời gian) và không quá ba phép quay cho phép xóa, hai cho phép chèn. Toàn bộ các giải thuật chèn và xóa có độ phức tạp thời gian cỡ O(log n).

## Phép chèn

Phép chèn bắt đầu bằng việc bổ sung một nút như trong cây tìm kiếm nhị phân bình thường và gán cho nó màu đỏ. Ta xem xét để bảo toàn tính chất đỏ đen từ các nút lân cận với nút mới bổ sung. Thuật ngữ nút chú bác sẽ dùng để chỉ nút anh (hoặc em) với nút cha của nút đó như trong cây phả hệ. Chú ý rằng:

* Tính chất 3 (Tất cả các lá -là các nút null là đen) giữ nguyên.
* Tính chất 4 (Cả hai con của nút đỏ là đen) nếu bị thay đổi chỉ bởi việc thêm một nút đỏ có thể sửa bằng cách gán màu đen cho một nút đỏ hoặc một phép quay.
* Tính chất 5 (Tất các các đường đi từ gôc tới các lá có cùng một số nút đen) nếu bị thay đổi chỉ bởi việc thêm một nút đỏ có thể sửa bằng cách gán màu đen cho một nút đỏ hoặc một phép quay.

Chú ý: Nhãn N sẽ dùng để chỉ nút đang chèn vào, P chỉ nút cha của N', G chỉ ông của N', và U chỉ chú bác của N'. Nhớ rằng,giữa các trường hợp, vai trò và nhãn của các nút có thể thay đổi còn trong cùng một trường hợp thì không.

Mỗi trường hợp được giới thiệu bằng một đoạn mã C. Nút chú bác và nút ông dễ dàng xác định nhờ các hàm sau:

struct node \*grandparent(struct node \*n) {

return n->parent->parent;

}

struct node \*uncle(struct node \*n) {

if (n->parent == grandparent(n)->left)

return grandparent(n)->right;

else

return grandparent(n)->left;

}

**Trường hợp 1:** Nút mới thêm N ở tại gốc. Trong trường hợp này, gán lại màu đen cho N, để bảo toàn tính chất 2 (Gốc là đen). Vì mới chỉ bổ sung một nút, Tính chất 5 được bảo đảm vì mọi đường đi chỉ có một nút.

void insert\_case1(struct node \*n) {

if (n->parent == NULL)

n->color = BLACK;

else

insert\_case2(n);

}

**Trường hợp 2:** Nút cha P của nút mới thêm là đen, khi đó Tính chất 4 (Cả hai nút con của nút đỏ là đen) không bị vi phạm vì nút mới thêm có hai con là "null' là đen. Tính chất 5 cũng không vi phạm vì nút mới thêm là đỏ không ẩnh hưởng tới số nút đen trên tất cả đường đi.

void insert\_case2(struct node \*n) {

if (n->parent->color == BLACK)

return; /\* Tree is still valid \*/

else

insert\_case3(n);

}

Chú ‎ý: Trong trường hợp tiếp theo nếu N có ông là nút G, vì nếu cha P là đỏ và P không ở gốc thì G là đen. Như vậy, N cũng có chú bác là U, although it may be a leaf in cases 4 and 5.

**Trường hợp 3:** Cả cha P và bác U là đỏ, thì thể đổi cả hai thành đen còn G thành đỏ (để bảo toàn tính chất 5 .Khi đó nút mới N có cha đen. Vì đường đi bất kỳ đi qua cha và bác của "N" phải đi qua ông của N nên số các nút đen trên đường đi này không thay đổi. Tuy thế nút ông G có thể vi phạm tính chất 2 (Gốc là đen) hoặc 4 (Cả hai con của nút đỏ là nút đen) (tính chất 4 bị vi phạm khi cha của G là đỏ). Để sửa chữa trường hợp này gọi một thủ tục đệ quy trên G từ trường hợp 1. Note that this is the only recursive call, and it occurs prior to any rotations, which proves that a constant number of rotations occur.

void insert\_case3(struct node \*n) {

if (uncle(n) != NULL && uncle(n)->color == RED) {

n->parent->color = BLACK;

uncle(n)->color = BLACK;

grandparent(n)->color = RED;

insert\_case1(grandparent(n));

}

else

insert\_case4(n);

}

Chú ‎ý: Trong các trường hợp tiếp theo, giả sử rằng nút cha P là con trái của cha của nó. Nếu nó là con phải, left và right đổi chỗ cho nhau trong cases 4 and 5.

**Trường hợp 4:** Nút cha P là đỏ nhưng nút chú bác U là đen, nút mới N là con phải của nút P, và P là con trái của nút G. Trong trường hợp này, thực hiện quay trái chuyển đổi vai trò của nút mới N và nút cha P do đó định dạng lại nút P bằng Trường hợp 5 (đổi vai trò N và P) vì tính chất 4 bị vi phạm (Cả hai con của nút đỏ là đen) . Phép quay cũng làm thay đổi một vài đường đi (các đường đi qua cây con nhãn "1") phải đi qua thêm nút mới N, nhưng vì N là đỏ nên không làm chúng vi pham tính chất 5

void insert\_case4(struct node \*n) {

if (n = = n->parent->right && n->parent = = grandparent(n)->left) {

rotate\_left(n->parent);

n = n->left;

} else if (n = = n->parent->left && n->parent = = grandparent(n)->right) {

rotate\_right(n->parent);

n = n->right;

}

insert\_case5(n);

}



**Trường hợp 5:**' Nút cha P là đỏ nhưng nút bác U là đen, nút mới N là con trái của nút P, và P là con trái của nút ông G. Trong trường hợp này, một phép quay phải trên nút ông G được thực hiện; kết quả của phep quay là trong cây mới nút P trở thành cha của cả hai nít N và nút G. Đã biết G là đen, vì bây giờ nó là con của P . Đổi màu của P và G thì cây thỏa mãn tính chất 4. Tính chất 5 không bị vi phạm vì các đường đi qua G trươc đây bây giờ đi qua P.

void insert\_case5(struct node \*n) {

n->parent->color = BLACK;

grandparent(n)->color = RED;

if (n == n->parent->left && n->parent == grandparent(n)->left) {

rotate\_right(grandparent(n));

} else {

/\* Here, n == n->parent->right && n->parent == grandparent(n)->right \*/

rotate\_left(grandparent(n));

}

}

## Phép xóa

Trong cây tìm kiếm nhị phân bình thường khi xóa một nút có cả hai con (không là lá null), ta tìm phần tử lớn nhất trong cây con trái hoặc phần tử nhỏ nhất trong cây co phải, chuyển giá trị của nó vào nút đang muốn xóa (xem Cây tìm kiếm nhị phân). Khi đó chúng ta xóa đi nút đã được copy giá trị, nút này có ít hơn hai con (không là lá null) . Vì việc copy giá trị không làm mất tính chất đỏ đen nên không cần phải sửa chữa gì cho thao tác này. Việc này chỉ đặt ra khi xóa các nút có nhiều nhất một con (không là lá null).

Chúng ta sẽ thảo luận về việc xóa một nút có nhiều nhất một con (không là lá null) .

Nếu ta xóa một nút đỏ, ta có thể chắc chắn rằng con của nó là nút đen. Tất cả các đường đi đi qua nút bị xóa chỉ đơn giản bớt đi một nút đỏ do đó tính chất 5 không thay đổi. Ngoài ra, cả nút cha và nút con của nút bị xóa đều là nút đen, do đó tính chất 3 và 4 vẫn giữa nguyên.. Một trường hợp đơn giản khác là khi xóa một nút đen chỉ có có một con là nút đỏ. Khi xóa nút đó các tính chất 4 và 5 bị phá vỡ, nhưng nếu gán lại màu cho nút con là đen thì chúng lại được khôi phục.

Trường hợp phức tạp xảy ra khi cả nút bị xóa và nút con của nó đều là đen. Chúng ta sẽ bắt đầu bằng việc thay nút bị xóa bằng nút con của nó. Chúng ta sẽ gọi nút con này (trong vị trí mới của nó là N, và anh em với nó (con khác của nút cha mới) là S. Tiếp theo ta vẫn dùng P chỉ cha mới của N, SL chỉ con trái của S, và SR chỉ con phải của S (chúng tồn tại vì S không thể là lá).

Chú ‎ý: Giữa các trường hợp khác nhau, vai trò và nhãn của các nút có thể thay đổi, nhưng trong một trường hợp mọi nhãn giữ vai trò không thay đổi. Trong hình vẽ các màu đỏ đen được thể hiện khi màu của nút đã rõ ràng, màu trắng biểu thị một màu chưa rõ (hoặc đỏ hoặc đen).

Chúng ta sẽ sử dụng hàm sau tìm người anh em của N':

struct node \*sibling(struct node \*n) {

if (n = = n->parent->left)

return n->parent->right;

else

return n->parent->left;

}

Chú ‎ý: Tất nhiên, chúng ta cần hoàn chỉnh các lá null sau mọi phép thay đổi. Nếu nút bị xóa không có con N khác "lá null", dễ dàng thấy rằng các tính chất được thỏa mãn. Còn nếu N là một "lá nulll", có thể sửa chữa lược đồ (hoặc code) để trong tất cảc các trường hợp các tính chất được thỏa mãn.

Trươc mỗi bước ta có thể dùng hàm (function) replace\_node thay thế nút con child vào vị trí của nút bị xóa trên cây. Để thuận tiện các đoạn code trong mục này quy ước rằng các "lá null" được biểu diễn bằng các đối tượng nút thực sự khác biệt một chút với NULL (code trong phép chèn có biểu diễn không nư vậy).

void delete\_one\_child(struct node \*n) {

/\* Giả thiết : n có ít nhất một nút con null \*/

struct node \*child = is\_leaf(n->right) ? n->left : n->right;

replace\_node(n, child);

if (n->color == BLACK) {

if (child->color == RED)

child->color = BLACK;

else

delete\_case1(child);

}

free(n);

}

Ghi chú: Nếu N là "lá nul" và ta khong muốn biểu diễn các "lá null" bằng các đối tượng nút thực, ta có thể sửa giải thuật bằng cách trước hết gọi delete\_case1() trên cha của nó (nghĩa là nút bị xóa n trong đoạn code trên) rồi sau đó mới xóa nó. Ta có thể làm như vậy vì cha của nó là đen, do đó có diễn biến như với "lá null" (một số người gọi là "lá ảo", "lá ma"). Ta cũng có thể xóa nó khỏi cuối của n và sẽ khôi phục lại sau tất cả các phép toán.

Nếu cả N và gốc ban đầu của nó là đen thì sau khi xóa các đường qua "N" giảm bớt một nút đen. Do đó vi phạm Tính chất 5, cây cần phải cân bằng lại. Có các trường hợp sau:

### Trường hợp 1

**Trường hợp 1:** N là gốc mới. Trong trường hợp này chúng ta dừng lại. Ta đã giải phóng một nút đen khỏi mọi đường đi và gôc mới lại là đen. Không tính chất nào bị vi phạm.

void delete\_case1(struct node \*n) {

if (n->parent == NULL)

return;

else

delete\_case2(n);

}

Chú ‎ý: Trong các trường hợp 2, 5, và 6, ta quy ước N là con trái của cha P. Nếu no là con phải, left và right sẽ tráo đổi cho nhau . Tuy nhiên code ví dụ làm cho cả hai trường hợp.

### Trường hợp 2



**Trường hợp 2:** S là đỏ. Trong trường hợp này tráo đổi màu của P và S, và sau đó quay trái tai P, nó sẽ làm cho S trở thành nút ông của N. Chú ý rằng P có màu đen và có một con màu đỏ. Tất cả các đường đi có số các nút đen giống nhau, bây giờ N có một anh em màu đen và cha màu đỏ, chúng ta có thể tiếp tục với các trường hợp 4, 5, hoặc 6. (anh em mới của nó là đen ví chỉ có một con của nút đỏ S.) Trong các trường hợp sau la sẽ gọi anh em mới của N' là S.

void delete\_case2(struct node \*n) {

if (sibling(n)->color == RED) {

n->parent->color = RED;

sibling(n)->color = BLACK;

if (n == n->parent->left)

rotate\_left(n->parent);

else

rotate\_right(n->parent);

}

delete\_case3(n);

}

### Trường hợp 3



**Trường hợp 3:** P, S, và các con của S là đen. Trong trường hợp này, chúng ta gán lại cho S màu đỏ. Kết quả là mọi đường đi qua S, (tất nhiên chúng không qua N,có ít hơn một nút đen. Vì việc xóa đi cha trước đây của N' làm tất cả các đương đi qua N bớt đi một nút đen, nên chúng bằng nhau. Tuy nhiên tất cả các đường đi qua P bây giờ có ít hơn một nút đen so với các đường không qua P, do đó Tính chất 5 (Tất cả các đường đi từ gốc tới các nút lá có cùng số nút đen) sẽ bị vi phạm. Để sửa chữa nó chúng ta lại tái cân bằng tại P, bắt đầu từ trường hợp 1.

void delete\_case3(struct node \*n) {

if (n->parent->color == BLACK &&

sibling(n)->color == BLACK &&

sibling(n)->left->color == BLACK &&

sibling(n)->right->color == BLACK)

{

sibling(n)->color = RED;

delete\_case1(n->parent);

}

else

delete\_case4(n);

}

### Trường hợp 4



**Trường hợp 4:** S và các con của S là đen nhưng P là đỏ. Trong trường hợp này, chúng ta đổi ngược màu của S và P. Điều này không ảnh hưởng tới số nút đen trên các đường đi không qua N, nhưng thêm một nút đen trên các đường đi qua N, thay cho nút đen đã bị xóa trên các đường này.

void delete\_case4(struct node \*n) {

if (n->parent->color == RED &&

sibling(n)->color == BLACK &&

sibling(n)->left->color == BLACK &&

sibling(n)->right->color == BLACK)

{

sibling(n)->color = RED;

n->parent->color = BLACK;

}

else

delete\_case5(n);

}

### Trường hợp 5



**Trường hợp 5:** S là đen, con trái của S là đỏ, con phải của S là đen, còn N là con trái của cha nó. Trong trường hợp này chúng ta quay phải tại S, khi đó con trái của S trở thành cha của S và N là anh em mới của nó. Sau đó ta tráo đổi màu của S và cha mới của nó. Tất cả các đường đi sẽ có số nút đen như nhau, nhưng bây giờ N có một người anh em đen mà con phải của nó lại là đỏ, chúng ta chuyển sang Trường hợp 6. Hoặc N hoặc cha của nó bị tác động bởi việc dịch chuyên này.

(Lưu ý trong trường hợp 6, ta đặt lại nút anh em mới của N là S.)

void delete\_case5(struct node \*n) {

if (n == n->parent->left &&

sibling(n)->color == BLACK &&

sibling(n)->left->color == RED &&

sibling(n)->right->color == BLACK)

{

sibling(n)->color = RED;

sibling(n)->left->color = BLACK;

rotate\_right(sibling(n));

}

else if (n == n->parent->right &&

sibling(n)->color == BLACK &&

sibling(n)->right->color == RED &&

sibling(n)->left->color == BLACK)

{

sibling(n)->color = RED;

sibling(n)->right->color = BLACK;

rotate\_left(sibling(n));

}

delete\_case6(n);

}

### Trường hợp 6



Trường hợp 6: S là đen, con phải của S là đỏ và N là con trái của nút cha P. Trong trường hợp này chúng ta quay trái tại P, khi đó S trở thành cha của P và con phải của S. Chúng ta hoán đổi màu của P và S, và gán cho con phải của S màu đen. Cây con giữ nguyên màu của gốc do đó Tính chất 4 (Cả hai con của nút đỏ là đen) và Tính chất 5 không bị vi phạm trong cây con này. Tuy nhiên, N bây giờ có thêm một nút đen tiền nhiệm: hoặc P mới bị tô đen, nó đã là đen và S là nút ông của nó trở thành đen. Như cậy các đương đi qua N có thêm một nút đen.

Trong lúc đó, với một đường đi không đi qua N, có hai khả năng:

* đi qua nút anh em của N. Khi đó cả trước và sau khi quay nó phải đi qua S và P, khi thay đổi màu sắc hai nút này đã tráo đổi màu cho nhau. Như vây đường đi này không bị thay đổi số nút đen.
* đi qua nút bác của N', là con phải của S. Khi đó trước khi quay nó đi qua S, cha của S, và con phải của S, nhưng sau khi quay nó chỉ đi qua nút S và con phải của S, khi này S đã nhận màu cũ của cha P còn con phải của S's đã đổi màu từ đỏ thành đen. Kết quả là số các nút đen trên đường đi này không thay đổi.

Như vậy, số các nút đen trên các đường đi là không thay đổi. Do đó các tính chất 4 và 5 đã được khôi phục. Nút trắng trong hình vẽ có thể là đỏ hoặc đen, nhưng phải ghi lại trước và sau khi thay đổi.

void delete\_case6(struct node \*n) {

sibling(n)->color = n->parent->color;

n->parent->color = BLACK;

if (n == n->parent->left) {

/\* Here, sibling(n)->right->color == RED \*/

sibling(n)->right->color = BLACK;

rotate\_left(n->parent);

}

else

{

/\* Here, sibling(n)->left->color == RED \*/

sibling(n)->left->color = BLACK;

rotate\_right(n->parent);

}

}

Nhắc lại rằng các hàm này có lời gọi đệ quy. Thêm nữa lời gọi không đệ quy sẽ được goi sau một phép quay, do đó số lần thực hiện các phép quay là không đổi (không quá 3).

## Câu 9: Thuật toán tìm kiếm theo chiều sâu trong đồ thị vô hướng[[1]](https://vi.wikipedia.org/wiki/T%C3%ACm_ki%E1%BA%BFm_theo_chi%E1%BB%81u_s%C3%A2u#cite_note-1)[[sửa](https://vi.wikipedia.org/w/index.php?title=T%C3%ACm_ki%E1%BA%BFm_theo_chi%E1%BB%81u_s%C3%A2u&veaction=edit&section=4) | [sửa mã nguồn](https://vi.wikipedia.org/w/index.php?title=T%C3%ACm_ki%E1%BA%BFm_theo_chi%E1%BB%81u_s%C3%A2u&action=edit&section=4)]

### Ý tưởng thuật toán[[sửa](https://vi.wikipedia.org/w/index.php?title=T%C3%ACm_ki%E1%BA%BFm_theo_chi%E1%BB%81u_s%C3%A2u&veaction=edit&section=5) | [sửa mã nguồn](https://vi.wikipedia.org/w/index.php?title=T%C3%ACm_ki%E1%BA%BFm_theo_chi%E1%BB%81u_s%C3%A2u&action=edit&section=5)]

1. **DFS** trên [đồ thị vô hướng](https://vi.wikipedia.org/wiki/Thu%E1%BA%ADt_ng%E1%BB%AF_l%C3%BD_thuy%E1%BA%BFt_%C4%91%E1%BB%93_th%E1%BB%8B#.C4.90) cũng giống như khám phá mê cung với một cuộn chỉ và một thùng sơn đỏ để đánh dấu, tránh bị lạc. Trong đó mỗi [đỉnh](https://vi.wikipedia.org/wiki/Thu%E1%BA%ADt_ng%E1%BB%AF_l%C3%BD_thuy%E1%BA%BFt_%C4%91%E1%BB%93_th%E1%BB%8B#.C4.90) **s** trong đồ thị tượng trưng cho một cửa trong mê cung.
2. Ta bắt đầu từ [đỉnh](https://vi.wikipedia.org/wiki/Thu%E1%BA%ADt_ng%E1%BB%AF_l%C3%BD_thuy%E1%BA%BFt_%C4%91%E1%BB%93_th%E1%BB%8B#.C4.90) **s**, buộc đầu cuộn chỉ vào **s** và đánh đấu [đỉnh](https://vi.wikipedia.org/wiki/Thu%E1%BA%ADt_ng%E1%BB%AF_l%C3%BD_thuy%E1%BA%BFt_%C4%91%E1%BB%93_th%E1%BB%8B#.C4.90) này **"đã thăm"**. Sau đó ta đánh dấu **s** là [đỉnh](https://vi.wikipedia.org/wiki/Thu%E1%BA%ADt_ng%E1%BB%AF_l%C3%BD_thuy%E1%BA%BFt_%C4%91%E1%BB%93_th%E1%BB%8B#.C4.90) hiện hành **u**.
3. Bây giờ, nếu ta đi theo [cạnh](https://vi.wikipedia.org/wiki/Thu%E1%BA%ADt_ng%E1%BB%AF_l%C3%BD_thuy%E1%BA%BFt_%C4%91%E1%BB%93_th%E1%BB%8B#.C) **(u,v)** bất kỳ.
4. Nếu [cạnh](https://vi.wikipedia.org/wiki/Thu%E1%BA%ADt_ng%E1%BB%AF_l%C3%BD_thuy%E1%BA%BFt_%C4%91%E1%BB%93_th%E1%BB%8B#.C) **(u,v)** dẫn chúng ta đến [đỉnh](https://vi.wikipedia.org/wiki/Thu%E1%BA%ADt_ng%E1%BB%AF_l%C3%BD_thuy%E1%BA%BFt_%C4%91%E1%BB%93_th%E1%BB%8B#.C4.90) **"đã thăm"** **v**, ta quay trở về **u**.
5. Nếu [đỉnh](https://vi.wikipedia.org/wiki/Thu%E1%BA%ADt_ng%E1%BB%AF_l%C3%BD_thuy%E1%BA%BFt_%C4%91%E1%BB%93_th%E1%BB%8B#.C4.90) **v** là [đỉnh](https://vi.wikipedia.org/wiki/Thu%E1%BA%ADt_ng%E1%BB%AF_l%C3%BD_thuy%E1%BA%BFt_%C4%91%E1%BB%93_th%E1%BB%8B#.C4.90) mới, ta di chuyển đến **v** và lăn cuộn chỉ theo. Đánh dấu **v** là **"đã thăm"**. Đặt **v** thành [đỉnh](https://vi.wikipedia.org/wiki/Thu%E1%BA%ADt_ng%E1%BB%AF_l%C3%BD_thuy%E1%BA%BFt_%C4%91%E1%BB%93_th%E1%BB%8B#.C4.90) hiện hành và lặp lại các bước.
6. Cuối cùng, ta có thể đi đến một [đỉnh](https://vi.wikipedia.org/wiki/Thu%E1%BA%ADt_ng%E1%BB%AF_l%C3%BD_thuy%E1%BA%BFt_%C4%91%E1%BB%93_th%E1%BB%8B#.C4.90) mà tại đó tất cả các cạnh kề với nó đều dẫn chúng ta đến các [đỉnh](https://vi.wikipedia.org/wiki/Thu%E1%BA%ADt_ng%E1%BB%AF_l%C3%BD_thuy%E1%BA%BFt_%C4%91%E1%BB%93_th%E1%BB%8B#.C4.90) **"đã thăm"**. Khi đó, ta sẽ quay lui bằng cách cuộn ngược cuộn chỉ và quay lại cho đến khi trở lại một [đỉnh kề](https://vi.wikipedia.org/wiki/Thu%E1%BA%ADt_ng%E1%BB%AF_l%C3%BD_thuy%E1%BA%BFt_%C4%91%E1%BB%93_th%E1%BB%8B#.C4.90) với một [cạnh](https://vi.wikipedia.org/wiki/Thu%E1%BA%ADt_ng%E1%BB%AF_l%C3%BD_thuy%E1%BA%BFt_%C4%91%E1%BB%93_th%E1%BB%8B#.C) còn chưa được khám phá. Lại tiếp tục quy trình khám phá như trên.
7. Khi chúng ta trở về **s** và không còn [cạnh](https://vi.wikipedia.org/wiki/Thu%E1%BA%ADt_ng%E1%BB%AF_l%C3%BD_thuy%E1%BA%BFt_%C4%91%E1%BB%93_th%E1%BB%8B#.C) nào kề với nó chưa bị khám phá là lúc **DFS** dừng.

### Mệnh đề[[sửa](https://vi.wikipedia.org/w/index.php?title=T%C3%ACm_ki%E1%BA%BFm_theo_chi%E1%BB%81u_s%C3%A2u&veaction=edit&section=6) | [sửa mã nguồn](https://vi.wikipedia.org/w/index.php?title=T%C3%ACm_ki%E1%BA%BFm_theo_chi%E1%BB%81u_s%C3%A2u&action=edit&section=6)]

Gọi **G** là một đồ thị vô hướng, trên đó ta sẽ thực hiện thao tác **DFS** với đỉnh bắt đầu là **s** thì:

1. Phép duyệt sẽ thăm tất cả các đỉnh cùng thành phần liên thông với **s**.
2. Các cạnh có nhãn **"đã thăm"** sẽ tạo ra một cây tối đại của thành phần liên thông chứa **s**.

### Chứng minh[[sửa](https://vi.wikipedia.org/w/index.php?title=T%C3%ACm_ki%E1%BA%BFm_theo_chi%E1%BB%81u_s%C3%A2u&veaction=edit&section=7) | [sửa mã nguồn](https://vi.wikipedia.org/w/index.php?title=T%C3%ACm_ki%E1%BA%BFm_theo_chi%E1%BB%81u_s%C3%A2u&action=edit&section=7)]

1. **Khẳng định 1**, là hiển nhiên vì **DFS** duyệt qua tất cả các đỉnh kề với đỉnh hiện hành(Có thể chứng minh hòa chỉnh hơn bằng phản chứng).
2. **Khẳng định 2**, đúng do ta chỉ đánh dấu các cạnh đi đến một đỉnh mới nên không thể tạo ra chu trình. Như vậy **DFS** tạo ra một [cây](https://vi.wikipedia.org/wiki/C%C3%A2y_(l%C3%BD_thuy%E1%BA%BFt_%C4%91%E1%BB%93_th%E1%BB%8B)). Hơn nữa, **DFS** thăm tất cả các đỉnh thuộc thành phần liên thông nên cây này là [cây tối đại](https://vi.wikipedia.org/wiki/C%C3%A2y_t%E1%BB%91i_%C4%91%E1%BA%A1i).

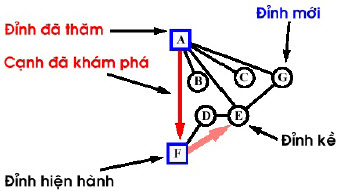
### Độ phức tạp của thuật toán[[sửa](https://vi.wikipedia.org/w/index.php?title=T%C3%ACm_ki%E1%BA%BFm_theo_chi%E1%BB%81u_s%C3%A2u&veaction=edit&section=8) | [sửa mã nguồn](https://vi.wikipedia.org/w/index.php?title=T%C3%ACm_ki%E1%BA%BFm_theo_chi%E1%BB%81u_s%C3%A2u&action=edit&section=8)]

1. **DFS** được gọi đúng **1** lần ứng với mỗi đỉnh.
2. Mỗi cạnh được xem xét đúng **2** lần, mỗi lần từ một đỉnh kề với nó.
3. Với **ns** đỉnh và **ms** cạnh thuộc thành phần liên thông chứa **s**, một phép **DFS** bắt đầu tại **s** sẽ chạy với thời gian **O(ns + ms)** nếu:

* Đồ thị được biểu diễn bằng cấu trúc dữ liệu dạng danh sách kề.
* Đặt nhãn cho một đỉnh là **"đã thăm"** và kiểm tra xem một đỉnh "đã thăm chưa tốn chi phí **O(degree)**.
* Bằng cách đặt nhãn cho các đỉnh là **"đã thăm"**, ta có thể xem xét một cách hệ thống các cạnh kề với đỉnh hiện hành nên ta sẽ không xem xét một cạnh quá 1 lần.

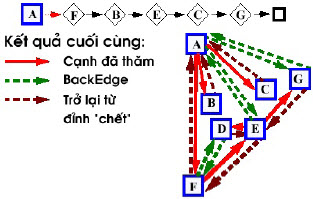
## Xác định đỉnh kề trong DFS[[sửa](https://vi.wikipedia.org/w/index.php?title=T%C3%ACm_ki%E1%BA%BFm_theo_chi%E1%BB%81u_s%C3%A2u&veaction=edit&section=9) | [sửa mã nguồn](https://vi.wikipedia.org/w/index.php?title=T%C3%ACm_ki%E1%BA%BFm_theo_chi%E1%BB%81u_s%C3%A2u&action=edit&section=9)]

* Kết quả của **DFS** phụ thuộc vào cách ta chọn đỉnh kế tiếp.

[](https://vi.wikipedia.org/wiki/T%E1%BA%ADp_tin:DFS1.jpg)

Xác định đỉnh kề trong Depth-first search

* Nếu ta bắt đầu tại **A** và thử cạnh nối đến **F**, sau đó đến **B**, rồi đến **E**, **C**, cuối cùng là **G** ta được:

[](https://vi.wikipedia.org/wiki/T%E1%BA%ADp_tin:DFS2.jpg)

Bắt đầu từ **A** và kết thúc tại **G**

* Nếu cũng bắt đầu từ **A** nhưng đi theo trình tự, tập các cạnh đã thăm, **backedge** và các điểm đệ quy sẽ khác trước.

[https://upload.wikimedia.org/wikipedia/commons/8/83/DFS3.jpg](https://vi.wikipedia.org/wiki/T%E1%BA%ADp_tin:DFS3.jpg)

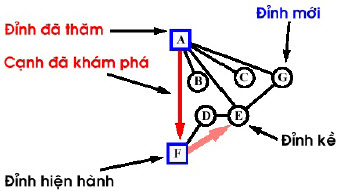
Bắt đầu từ **A** nhưng đi theo trình tự tập các cạnh đã thăm.

### Chạy từng bước thuật toán[[sửa](https://vi.wikipedia.org/w/index.php?title=T%C3%ACm_ki%E1%BA%BFm_theo_chi%E1%BB%81u_s%C3%A2u&veaction=edit&section=10) | [sửa mã nguồn](https://vi.wikipedia.org/w/index.php?title=T%C3%ACm_ki%E1%BA%BFm_theo_chi%E1%BB%81u_s%C3%A2u&action=edit&section=10)]

Giờ ta sẽ chạy từng bước thuật toán theo ví dụ trên.

#### Nguyên lý[**[2]**](https://vi.wikipedia.org/wiki/T%C3%ACm_ki%E1%BA%BFm_theo_chi%E1%BB%81u_s%C3%A2u#cite_note-2)**[**[**sửa**](https://vi.wikipedia.org/w/index.php?title=T%C3%ACm_ki%E1%BA%BFm_theo_chi%E1%BB%81u_s%C3%A2u&veaction=edit&section=11)**|**[**sửa mã nguồn**](https://vi.wikipedia.org/w/index.php?title=T%C3%ACm_ki%E1%BA%BFm_theo_chi%E1%BB%81u_s%C3%A2u&action=edit&section=11)**]**

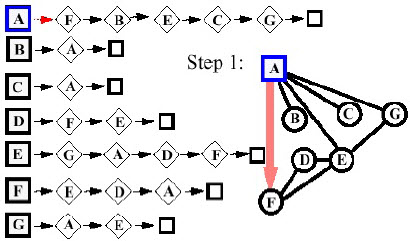
Khởi đầu từ một đỉnh, đi theo các cung(cạnh) xa nhất có thể. Trở lại đỉnh của cạnh xa nhất, tiếp tục duyệt như trước, cho đến đỉnh cuối cùng.

[](https://vi.wikipedia.org/wiki/T%E1%BA%ADp_tin:DFS1.jpg)

Thuật toán Depth-first search

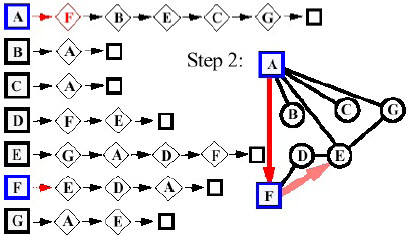
|  |  |
| --- | --- |
| [Nuvola apps kedit.png](https://vi.wikipedia.org/wiki/T%E1%BA%ADp_tin:Nuvola_apps_kedit.png) | **Đoạn này đang chờ cập nhật hướng dẫn chi tiết các bước chạy.** Bạn có thể [viết thêm cho đoạn này được hoàn thiện hơn](https://vi.wikipedia.org/w/index.php?title=T%C3%ACm_ki%E1%BA%BFm_theo_chi%E1%BB%81u_s%C3%A2u&action=edit). Xem phần [trợ giúp](https://vi.wikipedia.org/wiki/Tr%E1%BB%A3_gi%C3%BAp:S%E1%BB%ADa_%C4%91%E1%BB%95i) để biết thêm về cách sửa đổi bài. Nếu trang này [đã hoàn thiện](https://vi.wikipedia.org/w/index.php?title=T%C3%ACm_ki%E1%BA%BFm_theo_chi%E1%BB%81u_s%C3%A2u&action=history), mời bạn gỡ bản mẫu này. [Sửa đổi cuối:](https://vi.wikipedia.org/w/index.php?title=T%C3%ACm_ki%E1%BA%BFm_theo_chi%E1%BB%81u_s%C3%A2u&diff=cur) [Mai Ngọc Xuân](https://vi.wikipedia.org/wiki/Th%C3%A0nh_vi%C3%AAn:Mai_Ng%E1%BB%8Dc_Xu%C3%A2n) ([thảo luận](https://vi.wikipedia.org/wiki/Th%E1%BA%A3o_lu%E1%BA%ADn_Th%C3%A0nh_vi%C3%AAn:Mai_Ng%E1%BB%8Dc_Xu%C3%A2n) **·** [đóng góp](https://vi.wikipedia.org/wiki/%C4%90%E1%BA%B7c_bi%E1%BB%87t:%C4%90%C3%B3ng_g%C3%B3p/Mai_Ng%E1%BB%8Dc_Xu%C3%A2n)) vào 6 tháng trước. *(*[*làm mới*](https://vi.wikipedia.org/w/index.php?title=T%C3%ACm_ki%E1%BA%BFm_theo_chi%E1%BB%81u_s%C3%A2u&action=purge)*)* |

* Bước 1:

[](https://vi.wikipedia.org/wiki/T%E1%BA%ADp_tin:DFS-Step1.jpg)

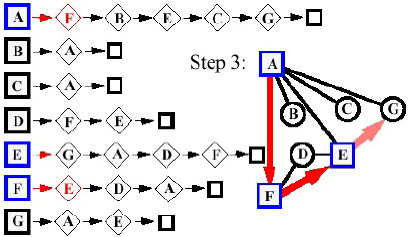
Chạy từng bước thuật toán Depth-first search trong đồ thị vô hướng, bước 1

* Bước 2:

[](https://vi.wikipedia.org/wiki/T%E1%BA%ADp_tin:DFS-Step2.jpg)

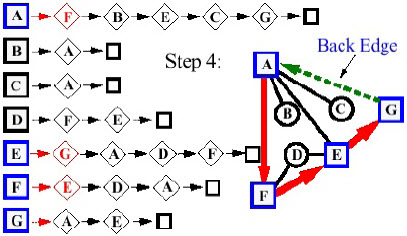
Chạy từng bước thuật toán Depth-first search trong đồ thị vô hướng, bước 2

* Bước 3:

[](https://vi.wikipedia.org/wiki/T%E1%BA%ADp_tin:DFS-Step3.jpg)

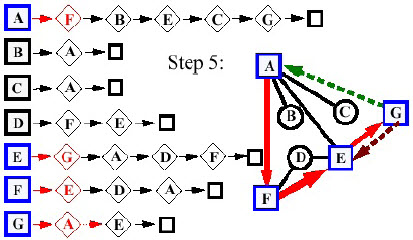
Chạy từng bước thuật toán Depth-first search trong đồ thị vô hướng, bước 3

* Bước 4:

[](https://vi.wikipedia.org/wiki/T%E1%BA%ADp_tin:DFS-Step4.jpg)

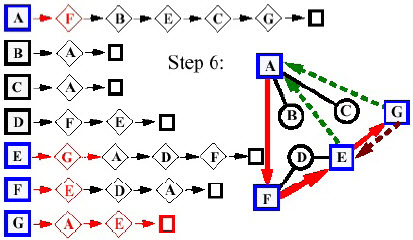
Chạy từng bước thuật toán Depth-first search trong đồ thị vô hướng, bước 4

* Bước 5:

[](https://vi.wikipedia.org/wiki/T%E1%BA%ADp_tin:DFS-Step5.jpg)

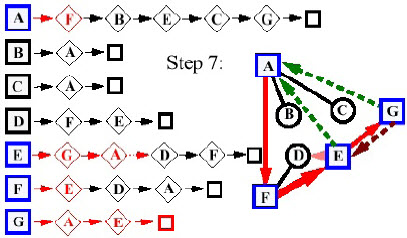
Chạy từng bước thuật toán Depth-first search trong đồ thị vô hướng, bước 5

* Bước 6:

[](https://vi.wikipedia.org/wiki/T%E1%BA%ADp_tin:DFS-Step6.jpg)

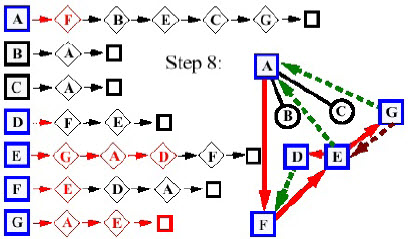
Chạy từng bước thuật toán Depth-first search trong đồ thị vô hướng, bước 6

* Bước 7:

[](https://vi.wikipedia.org/wiki/T%E1%BA%ADp_tin:DFS-Step7.jpg)

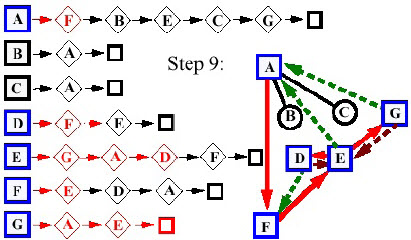
Chạy từng bước thuật toán Depth-first search trong đồ thị vô hướng, bước 7

* Bước 8:

[](https://vi.wikipedia.org/wiki/T%E1%BA%ADp_tin:DFS-Step8.jpg)

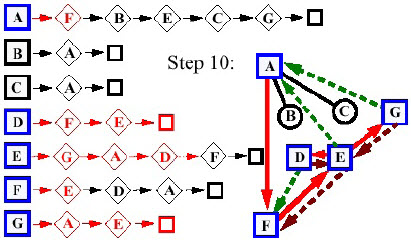
Chạy từng bước thuật toán Depth-first search trong đồ thị vô hướng, bước 8

* Bước 9:

[](https://vi.wikipedia.org/wiki/T%E1%BA%ADp_tin:DFS-Step9.jpg)

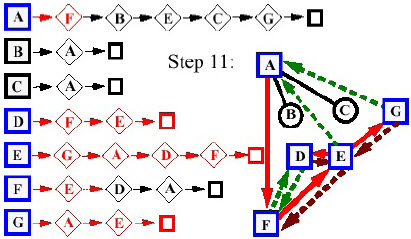
Chạy từng bước thuật toán Depth-first search trong đồ thị vô hướng, bước 9

* Bước 10:

[](https://vi.wikipedia.org/wiki/T%E1%BA%ADp_tin:DFS-Step10.jpg)

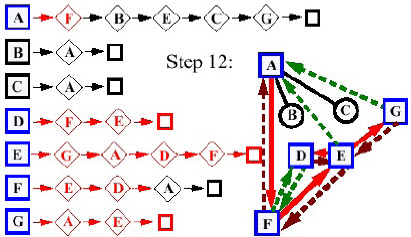
Chạy từng bước thuật toán Depth-first search trong đồ thị vô hướng, bước 10

* Bước 11:

[](https://vi.wikipedia.org/wiki/T%E1%BA%ADp_tin:DFS-Step11.jpg)

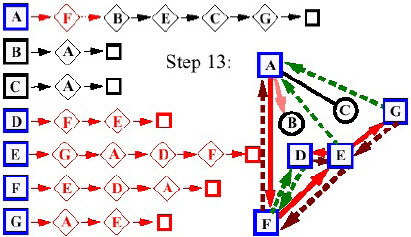
Chạy từng bước thuật toán Depth-first search trong đồ thị vô hướng, bước 11

* Bước 12:

[](https://vi.wikipedia.org/wiki/T%E1%BA%ADp_tin:DFS-Step12.jpg)

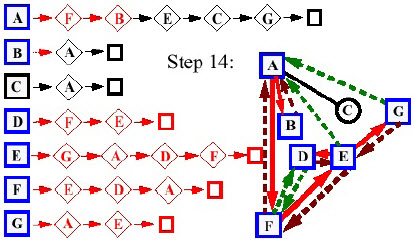
Chạy từng bước thuật toán Depth-first search trong đồ thị vô hướng, bước 12

* Bước 13:

[](https://vi.wikipedia.org/wiki/T%E1%BA%ADp_tin:DFS-Step13.jpg)

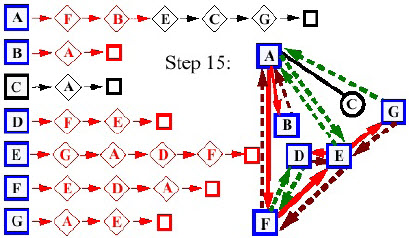
Chạy từng bước thuật toán Depth-first search trong đồ thị vô hướng, bước 13

* Bước 14:

[](https://vi.wikipedia.org/wiki/T%E1%BA%ADp_tin:DFS-Step14.jpg)

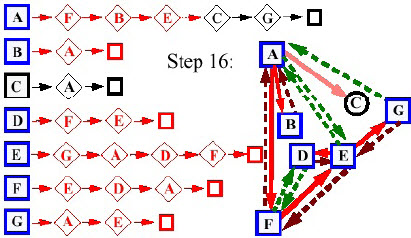
Chạy từng bước thuật toán Depth-first search trong đồ thị vô hướng, bước 14

* Bước 15:

[](https://vi.wikipedia.org/wiki/T%E1%BA%ADp_tin:DFS-Step15.jpg)

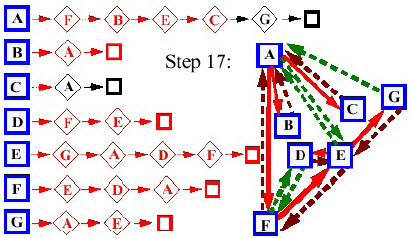
Chạy từng bước thuật toán Depth-first search trong đồ thị vô hướng, bước 15

* Bước 16:

[](https://vi.wikipedia.org/wiki/T%E1%BA%ADp_tin:DFS-Step16.jpg)

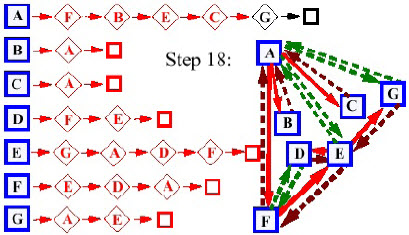
Chạy từng bước thuật toán Depth-first search trong đồ thị vô hướng, bước 16

* Bước 17:

[](https://vi.wikipedia.org/wiki/T%E1%BA%ADp_tin:DFS-Step17.jpg)

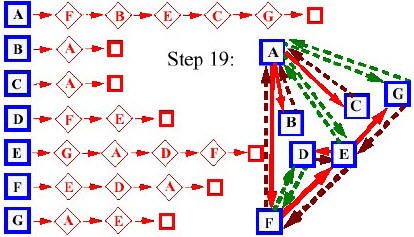
Chạy từng bước thuật toán Depth-first search trong đồ thị vô hướng, bước 17

* Bước 18:

[](https://vi.wikipedia.org/wiki/T%E1%BA%ADp_tin:DFS-Step18.jpg)

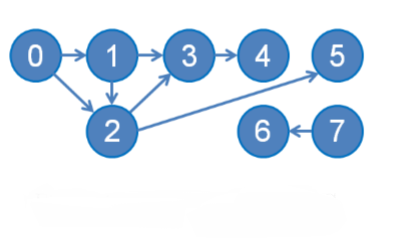
Chạy từng bước thuật toán Depth-first search trong đồ thị vô hướng, bước 18

* Bước 19:

[](https://vi.wikipedia.org/wiki/T%E1%BA%ADp_tin:DFS-Step19.jpg)

Chạy từng bước thuật toán Depth-first search trong đồ thị vô hướng, bước 19

Câu 11: Trong khoa học máy tính, thứ tự tô pô của một đồ thị có hướng là một thứ tự sắp xếp của các đỉnh sao cho với mọi cung từ u đến v trong đồ thị, u luôn nằm trước v. Thuật toán để tìm thứ tự tô pô gọi là **thuật toán sắp xếp tô pô**. Thứ tự tô pô tồn tại khi và chỉ khi **đồ thị không có chu trình** **(viết tắt là DAG - tiếng Anh directed acyclic graph)**. Đồ thị có hướng không có chu trình luôn có ít nhất một thứ tự tô pô, và có thuật toán để tìm thứ tự tô pô trong thời gian tuyến tính... ([Wikipedia](https://vi.wikipedia.org/wiki/S%E1%BA%AFp_x%E1%BA%BFp_t%C3%B4_p%C3%B4)).

[](https://sites.google.com/site/nhatnguyendrgs/home/programming/graph/topo-sort/Figure%204.4.png?attredirects=0)

**Input:**

8 8

0 1 0

0 2 0

1 2 0

1 3 0

2 3 0

2 5 0

3 4 0

**Output:**

Topological Sort (the input graph must be DAG)

7 6 0 1 2 5 3 4

**Code (tham khảo Competitive Programming):**

// nhatnguyendrgs (c) 2015 - toposort.cpp

#include "iostream"

#include "stdio.h"

#include "stdlib.h"

#include "string"

#include "string.h"

#include "algorithm"

#include "math.h"

#include "vector"

#include "map"

#include "queue"

#include "stack"

#include "deque"

#include "set"

using namespace std;

typedef pair<int, int> ii;

typedef vector<int> vi;

typedef vector<ii> vii;

const int inf = 1e9;

#define DFS\_WHITE -1 // UNVISITED

#define DFS\_BLACK 1 // EXPLORED

#define DFS\_GRAY 2 // VISTED BUT NOT EXPLORED

vector<vii> AdjList;

int V, E, u, v, w;

void graphDirected(){

scanf("%d %d", &V, &E);

AdjList.assign(V + 4, vii());

for (int i = 0; i < E; i++) {

scanf("%d %d %d", &u, &v, &w);

AdjList[u].push\_back(ii(v, w));

}

}

vi dfs\_num;

vi topoSort; // global vector to store the toposort in reverse order

bool cycle = false; // check cycle in graph

void topo(int u) { // change function name to differentiate with original dfs

dfs\_num[u] = DFS\_GRAY;

for (int j = 0; j < AdjList[u].size(); j++){

ii v = AdjList[u][j];

if (dfs\_num[v.first] == DFS\_WHITE)

topo(v.first);

else if (dfs\_num[v.first] == DFS\_GRAY)

cycle = true;

}

topoSort.push\_back(u);

dfs\_num[u] = DFS\_BLACK;

}

int main(){

printf("Topological Sort (the input graph must be DAG)\n");

graphDirected();

topoSort.clear();

dfs\_num.assign(V+4, DFS\_WHITE);

for (int i = 0; i < V; i++) // this part is the same as finding CCs

if (dfs\_num[i] == DFS\_WHITE)

topo(i);

if (cycle == true)

printf(" Exist cycle.\n");

else{

reverse(topoSort.begin(), topoSort.end()); // reverse topoSort

for (int i = 0; i < (int)topoSort.size(); i++) // or you can simply read

printf(" %d", topoSort[i]); // the content of `topoSort' backwards

}

printf("\n");

return 0;

}

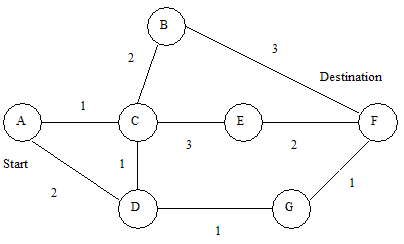
Câu 12: Cho 1 đồ thị có hướng **G = (V, E)** với các cạnh có trọng số không âm, có dữ liệu nhập vào là ma trận trọng số **L** và 2 đỉnh **x, y** cho trước. Việc ta cần làm là tìm đường đi ngắn nhất từ **x** đến **y** trong đồ thị **G**.

Việc chúng ta cần làm là chỉ ra đỉnh **v** bất kì sao cho **x -> v** là đường đi ngắn nhất. Ta gọi **length[v]** là giá trị đường đi ngắn nhất từ **x -> v**, có thể hiểu **length[v]** là giá trị đường đi ngắn nhất trong các đường đi từ đỉnh **x** qua các đỉnh trong tập hợp S (nếu có) rồi đến **v**.

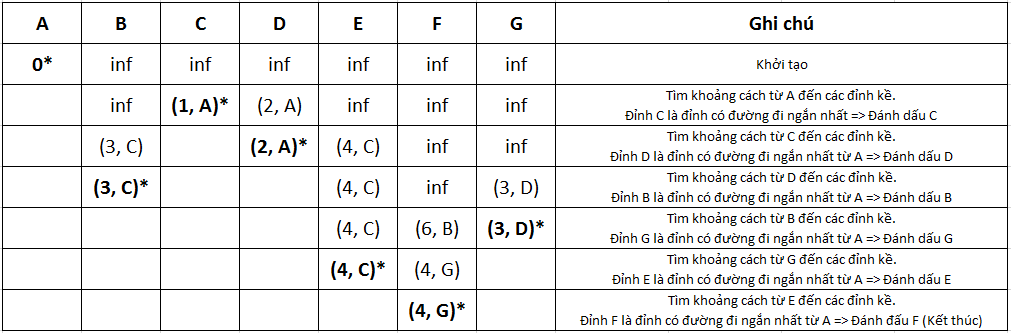
Thuật toán:

1. Khởi tạo các mảng **n**phần tử: **label, length, prev**. Gán **label[k] = 1, length[k] = -1 (inf), prev[k] = -1** với **k** chạy từ **0 -> n – 1**. Gán **length[first] = 0**
2. Chọn đỉnh **v** trong mảng sao cho **length[k]** là nhỏ nhất. Sau đó gán **label[k] = 0**(Đã đánh dấu)
3. Tạo vòng lặp với biến chạy **k**, xét nếu **label[k] = 1**(Chưa đánh dấu) và có đường đi từ **v -> k**: Nếu **length[k] > length[v] +**trọng số từ **v -> k** hoặc **length[k] = inf**, có nghĩa là nếu ta tìm được 1 đường từ **v -> k** là nhỏ nhất, hoặc là chưa tìm được đường nào ngắn nhất (inf) => Gán **length[k] = length[v] +**trọng số **v -> k**, **prev[k] = v**(Tạo vết chân đỉnh trước đó).
4. Nếu **label[last] = 0** (Đã đánh dấu đỉnh đến), kết thúc vòng lặp. Nếu không thì quay lại bước 2.

VD: Ta có 1 đồ thị như sau

[](http://trachanhso.net/wp-content/uploads/2015/11/dijkstras-graph.gif)Đồ thị đã cho.

Ta cần chỉ ra đường đi ngắn nhất từ đỉnh **A** tới **F**. Vậy các bước sẽ như thế nào?

[](http://trachanhso.net/wp-content/uploads/2015/11/record-dijkstra.png)Bảng thống kê các bước thực hiện.

Về nguyên lý, thuật toán Dijkstra giống như việc chạy thi đua vậy. Từ điểm đến ban đầu, ta sẽ mỗi người chạy đến điểm kết thúc theo các đường đi khác nhau. Nếu người nào chạy tới đích trước (tìm min và đánh dấu điểm kết thúc sớm nhất) thì ta xuất ra các đỉnh mà người đó đã đi qua.

**Câu 13:** Trong [khoa học máy tính](https://vi.wikipedia.org/wiki/Khoa_h%E1%BB%8Dc_m%C3%A1y_t%C3%ADnh), **thuật toán Kosaraju-Sharir** là một thuật toán tìm [thành phần liên thông mạnh](https://vi.wikipedia.org/wiki/Th%C3%A0nh_ph%E1%BA%A7n_li%C3%AAn_th%C3%B4ng_m%E1%BA%A1nh) trong đồ thị có hướng. Theo Aho, [Hopcroft](https://vi.wikipedia.org/w/index.php?title=John_E._Hopcroft&action=edit&redlink=1) và [Ullman](https://vi.wikipedia.org/w/index.php?title=Jeffrey_D._Ullman&action=edit&redlink=1)[[1]](https://vi.wikipedia.org/wiki/Thu%E1%BA%ADt_to%C3%A1n_Kosaraju#cite_note-1), thuật toán này xuất hiện trong một bài báo chưa được công bố năm 1978 của [S. Rao Kosaraju](https://vi.wikipedia.org/w/index.php?title=S._Rao_Kosaraju&action=edit&redlink=1) và [Micha Sharir](https://vi.wikipedia.org/w/index.php?title=Micha_Sharir&action=edit&redlink=1). Thuật toán sử dụng tính chất sau: trong đồ thị chuyển vị (đồ thị trong đó mọi cung được đảo ngược so với đồ thị ban đầu), các thành phần liên thông mạnh là không đổi so với đồ thị ban đầu.

Thuật toán Kosaraju-Sharir hoạt động như sau:

* Cho trước *G* là một đồ thị có hướng và một [ngăn xếp](https://vi.wikipedia.org/wiki/Ng%C4%83n_x%E1%BA%BFp) rỗng *S*.
* Chừng nào *S* còn chưa chứa tất cả các đỉnh trong *G*:
  + Chọn một đỉnh *v* bất kì không nằm trong *S*. Thực hiện [tìm kiếm theo chiều sâu](https://vi.wikipedia.org/wiki/T%C3%ACm_ki%E1%BA%BFm_theo_chi%E1%BB%81u_s%C3%A2u) bắt đầu từ *v*. Mỗi lần thuật toán tìm kiếm kết thúc việc tìm đường từ một đỉnh *u*, chèn *u* vào *S*.
* Đảo ngược tất cả các cung của đồ thị để có đồ thị chuyển vị
* Chừng nào *S* còn khác rỗng:
  + Lấy ra phần tử trên cùng *v* của *S*. Thực hiện tìm kiếm theo chiều sâu từ *v* trong đồ thị chuyển vị. Tập hợp các đỉnh được thăm chính là thành phần liên thông mạnh chứa *v*. Ghi nhận các đỉnh này và xóa chúng khỏi đồ thị và ngăn xếp *S*.

Độ phức tạp tính toán[[sửa](https://vi.wikipedia.org/w/index.php?title=Thu%E1%BA%ADt_to%C3%A1n_Kosaraju&veaction=edit&section=1) | [sửa mã nguồn](https://vi.wikipedia.org/w/index.php?title=Thu%E1%BA%ADt_to%C3%A1n_Kosaraju&action=edit&section=1)]

Giả sử đồ thị được biểu diễn dưới dạng [danh sách kề](https://vi.wikipedia.org/wiki/Danh_s%C3%A1ch_k%E1%BB%81), thuật toán Kosaraju-Sharir duyệt qua toàn bộ đồ thị hai lần, do đó có độ phức tạp tính toán tuyến tính Θ(V+E). Độ phức tạp này là tối ưu bởi mọi thuật toán đều phải xem xét tất cả các đỉnh và cung của đồ thị. Thuật toán này vô cùng đơn giản nhưng trong thực tế không hiệu quả bằng các thuật toán của Tarjan và Gabow, do chúng chỉ duyệt qua đồ thị đúng một lần.

Nếu đồ thị được biểu diễn dưới dạng [ma trận kề](https://vi.wikipedia.org/wiki/Ma_tr%E1%BA%ADn_k%E1%BB%81), thuật toán chạy trong thời gian Ο(V2)

## Câu 14: Thuật toán[[sửa](https://vi.wikipedia.org/w/index.php?title=Thu%E1%BA%ADt_to%C3%A1n_Karger&veaction=edit&section=1) | [sửa mã nguồn](https://vi.wikipedia.org/w/index.php?title=Thu%E1%BA%ADt_to%C3%A1n_Karger&action=edit&section=1)]

Ý tưởng chính của thuật toán là sử dụng phép hợp nhất hai đầu của một cạnh {\displaystyle e} trong đồ thị {\displaystyle G=(V,E)}. Sau mỗi lần hợp nhất, số đỉnh của đồ thị giảm đi 1. Thuật toán sử dụng một chuỗi các phép hợp nhất các cạnh ngẫu nhiên của đồ thị. Xác suất chọn mỗi cạnh tỉ lệ với trọng số của nó. Thuật toán này là một thuật toán đệ quy. Trong mỗi tầng đệ quy, thuật toán hoạt động như sau. Thử hai lần độc lập nhau việc lặp đi lặp lại phép hợp nhất để giảm số đỉnh của {\displaystyle G} xuống {\displaystyle \left\lceil n/{\sqrt {2}}+1\right\rceil } và gọi đệ quy để tính lát cắt nhỏ nhất trong đồ thị thu được. Sau đó, chọn kết quả tốt hơn trong hai lần gọi đệ quy và trả về giá trị đó.

## Phép hợp nhất[[sửa](https://vi.wikipedia.org/w/index.php?title=Thu%E1%BA%ADt_to%C3%A1n_Karger&veaction=edit&section=2) | [sửa mã nguồn](https://vi.wikipedia.org/w/index.php?title=Thu%E1%BA%ADt_to%C3%A1n_Karger&action=edit&section=2)]

Trong mỗi lần thực hiện, phép toán này hợp nhất hai đỉnh *x* và *y* của một cung *e* thành một đỉnh mới {\displaystyle v\_{e}} kề với tất cả các đỉnh kề của *x* và *y*. Định nghĩa cụ thể là như sau.

Cho một đồ thị {\displaystyle G=\left(V,E\right)} và {\displaystyle e=\lbrace x,y\rbrace \in E}, kết quả phép hợp nhất hai đỉnh kề với cạnh {\displaystyle e} (ký hiệu {\displaystyle G/e=\left(V',E'\right)}) là một [đa đồ thị](https://vi.wikipedia.org/wiki/%C4%90a_%C4%91%E1%BB%93_th%E1%BB%8B) định nghĩa như sau:

{\displaystyle V'=\left(V\setminus \lbrace x,y\rbrace \right)\cup \lbrace v\_{e}\rbrace }

và:

{\displaystyle E'=\lbrace \lbrace v,w\rbrace \in E\mid \lbrace v,w\rbrace \cap \lbrace x,y\rbrace =\emptyset \rbrace \cup \lbrace \lbrace v\_{e},w\rbrace \mid \lbrace x,w\rbrace \in E\setminus \lbrace e\rbrace } hoặc {\displaystyle \lbrace y,w\rbrace \in E\setminus \lbrace e\rbrace \rbrace }

Có thể chứng minh phép toán này không làm giảm nhưng có thể làm tăng giá trị lát cắt nhỏ nhất.

## Thời gian thực hiện[[sửa](https://vi.wikipedia.org/w/index.php?title=Thu%E1%BA%ADt_to%C3%A1n_Karger&veaction=edit&section=3) | [sửa mã nguồn](https://vi.wikipedia.org/w/index.php?title=Thu%E1%BA%ADt_to%C3%A1n_Karger&action=edit&section=3)]

Thuật toán Karger là thuật toán ngẫu nhiên nhanh nhất hiện nay cho việc tìm lát cắt nhỏ nhất, với thời gian chạy *O*(|*V*|2 log3|*V*|). Để chứng minh điều này, tác giả chỉ ra cách thực hiện chuỗi các phép hợp nhất để giảm kích thước {\displaystyle G} xuống {\displaystyle \left\lceil n/{\sqrt {2}}+1\right\rceil } đỉnh trong thời gian {\displaystyle O(n^{2})}. Do đó thời gian chạy của thuật toán là {\displaystyle T(n)=2\left(n^{2}+T\left(\left\lceil n/{\sqrt {2}}+1\right\rceil \right)\right)}. Phương trình này cho kết quả {\displaystyle T(n)=O(n^{2}log(n))}. Sau mỗi lần thực hiện thuật toán, xác suất tìm ra lát cắt nhỏ nhất là {\displaystyle \Omega (1/log(n))}. Nếu thực hiện thuật toán {\displaystyle O(log^{2}(n))} lần thì xác suất không tìm ra lát cắt nhỏ nhất giảm xuống O(1/*n*2).

## Câu 16: Định nghĩa chính thức [ [sửa](https://en.wikipedia.org/w/index.php?title=Activity_selection_problem&action=edit&section=1)]

Giả sử có tồn tại *n* hoạt động với mỗi người trong số họ được đại diện bởi một thời gian bắt đầu *của tôi* và kết thúc thời gian *f i* . Hai hoạt động *i* và *j* được cho là không mâu thuẫn nếu *s i* ≥ *f j* hoặc *s j* ≥ *f i* . Vấn đề lựa chọn hoạt động bao gồm việc tìm ra bộ giải pháp tối đa (S) của các hoạt động không mâu thuẫn, hoặc chính xác hơn phải không có [giải pháp thiết lập](https://en.wikipedia.org/wiki/Solution_set) S 'sao cho | S' | > | S | trong trường hợp nhiều giải pháp tối đa có kích cỡ bằng nhau.

## Giải pháp tối ưu [ [sửa](https://en.wikipedia.org/w/index.php?title=Activity_selection_problem&action=edit&section=2)]

Vấn đề lựa chọn hoạt động là đáng chú ý trong việc sử dụng một [thuật toán tham lam](https://en.wikipedia.org/wiki/Greedy_algorithm) để tìm một giải pháp luôn luôn sẽ dẫn đến một [giải pháp tối ưu](https://en.wikipedia.org/wiki/Optimal_solution) . Một phác thảo [giả mạo](https://en.wikipedia.org/wiki/Pseudocode) của phiên bản lặp của thuật toán và một bằng chứng về sự tối ưu của kết quả của nó được bao gồm bên dưới.

### Thuật toán [ [sửa](https://en.wikipedia.org/w/index.php?title=Activity_selection_problem&action=edit&section=3)]

1 tham lam - Lặp đi lặp lại - Hoạt động - Selector ( Một , s , f ) :

2

3 Sắp xếp Một bởi kết thúc thời gian lưu trữ trong f '

4

5 S = { A [ 1 ]}

6 k = 1

7

8 n = A . chiều dài

9

10 **đối với**  i = 2 đến n :

11 **nếu**  s [ i ] ≥ f [ k ] :

12 S = S U { A [ i ]}

13 k = i

14

15 **trả lại**  S

#### Giải thích **[**[**sửa**](https://en.wikipedia.org/w/index.php?title=Activity_selection_problem&action=edit&section=4)**]**

**Dòng 1:** Thuật toán này được gọi là Công cụ *Lựa chọn Hoạt động Nghèo đói* , bởi vì trước tiên nó là một thuật toán tham lam, và sau đó lặp đi lặp lại. Cũng có một phiên bản đệ quy của thuật toán tham lam này.

* {\ displaystyle A}là một mảng chứa các *hoạt động* .
* {\ displaystyle s}là mảng có chứa *thời gian bắt đầu* của các hoạt động trong{\ displaystyle A}.
* {\ displaystyle f}là một mảng có chứa *thời gian hoàn thành* của các hoạt động trong{\ displaystyle A}.

Lưu ý rằng các mảng này được lập chỉ mục bắt đầu từ 1 đến chiều dài của mảng tương ứng.

**Dòng 3:** Sắp xếp theo *thứ tự ngày càng tăng của thời gian kết thúc* mảng hoạt động{\ displaystyle A} bằng cách sử dụng thời gian kết thúc được lưu trữ trong mảng {\ displaystyle f}. Hoạt động này có thể được thực hiện trong{\ displaystyle O (n \ cdot \ log \_ {2} n)} thời gian, sử dụng ví dụ sắp xếp hợp nhất, sắp xếp heap, hoặc các thuật toán sắp xếp nhanh.

**Dòng 5:** Tạo một tập hợp{\ displaystyle S}để lưu trữ các *hoạt động đã chọn* và khởi tạo nó bằng hoạt động{\ displaystyle A [1]} đó là thời gian kết thúc sớm nhất.

**Dòng 6:** Tạo ra một biến{\ displaystyle k} theo dõi chỉ mục của hoạt động đã chọn cuối cùng.

**Dòng 10:** Bắt đầu lặp đi lặp lại từ phần tử thứ hai của mảng đó{\ displaystyle A} đến yếu tố cuối cùng của nó.

**Dòng 11,12:** Nếu *thời gian bắt đầu* {\ displaystyle s [i]} của {\ displaystyle ith} Hoạt động ({\ displaystyle A [i]}) lớn hơn hoặc bằng *thời gian kết thúc* {\ displaystyle f [k]}của *hoạt động đã chọn cuối cùng* ({\ displaystyle A [k]}), sau đó {\ displaystyle A [i]} tương thích với các hoạt động đã chọn trong tập hợp {\ displaystyle S}, và do đó nó có thể được thêm vào {\ displaystyle S}.

**Dòng 13:** Chỉ mục của hoạt động đã chọn cuối cùng được cập nhật cho hoạt động được thêm vào{\ displaystyle A [i]}.

### Bằng chứng về sự tối ưu [ [sửa](https://en.wikipedia.org/w/index.php?title=Activity_selection_problem&action=edit&section=5)]

Để cho {\ displaystyle S = \ {1,2, \ ldots, n \}}là bộ hoạt động được sắp xếp theo thời gian kết thúc. Giả sử{\ displaystyle A \ subseteq S}là một giải pháp tối ưu, cũng được sắp xếp theo thời gian kết thúc; và chỉ số của hoạt động đầu tiên trong *A* là{\ displaystyle k \ neq 1}, tức là, giải pháp tối ưu *này không* bắt đầu với sự lựa chọn tham lam. Chúng tôi sẽ chỉ ra rằng{\ displaystyle B = (A \ setminus \ {k \}) \ cup \ {1 \}}, bắt đầu với sự lựa chọn tham lam (hoạt động 1), là một giải pháp tối ưu. Kể từ{\ displaystyle f\_ {1} \ leq f\_ {k}}, và các hoạt động ở A [không liên quan](https://en.wikipedia.org/wiki/Disjoint_sets) theo định nghĩa, các hoạt động ở B cũng không liên kết. Vì *B* có cùng một số hoạt động như *A* , có nghĩa là,{\ displaystyle | A | = | B |}, *B* cũng là tối ưu.

Một khi sự lựa chọn tham lam được thực hiện, vấn đề giảm xuống để tìm ra một giải pháp tối ưu cho bài toán con. Nếu *A* là một giải pháp tối ưu cho vấn đề ban đầu *S* , sau đó{\ displaystyle A ^ {\ prime} = A \ setminus \ {1 \}} là một giải pháp tối ưu cho vấn đề lựa chọn hoạt động {\ displaystyle S '= \ {i \ in S: s\_ {i} \ geq f\_ {1} \}}.

Tại sao? Nếu chúng ta có thể tìm ra một giải pháp *B* 'sang *S* ' với nhiều hoạt động hơn thì *A* ', thêm 1 vào *B* ' sẽ mang lại một giải pháp *B* cho *S* với nhiều hoạt động hơn *A* , mâu thuẫn với sự tối ưu.

### Vấn đề lựa chọn hoạt động có trọng số [ [sửa](https://en.wikipedia.org/w/index.php?title=Activity_selection_problem&action=edit&section=6)]

Phiên bản tổng quát của vấn đề lựa chọn hoạt động liên quan đến việc lựa chọn một bộ tối ưu các hoạt động không chồng chéo sao cho tổng trọng lượng được tối đa hóa. Không giống như phiên bản không có trọng tải, không có giải pháp tham lam cho vấn đề lựa chọn hoạt động có trọng số. Tuy nhiên, một giải pháp [lập trình động](https://en.wikipedia.org/wiki/Dynamic_programming) có thể dễ dàng được hình thành bằng cách sử dụng cách tiếp cận sau: [[1]](https://en.wikipedia.org/wiki/Activity_selection_problem#cite_note-1)

Xem xét một giải pháp tối ưu có chứa hoạt động *k* . Bây giờ chúng ta có các hoạt động không chồng chéo ở bên trái và phải của *k* . Chúng ta có thể đệ quy tìm giải pháp cho hai bộ này vì cấu trúc con tối ưu. Như chúng ta không biết *k* , chúng ta có thể thử từng hoạt động. Cách tiếp cận này dẫn đến một{\ displaystyle O (n ^ {3})}dung dịch. Điều này có thể được tối ưu hoá hơn nữa khi xem xét cho mỗi bộ hoạt động trong{\ displaystyle (i, j)}, chúng ta có thể tìm ra giải pháp tối ưu nếu chúng ta biết giải pháp cho {\ displaystyle (i, t)}, trong đó *t* là khoảng cuối không chồng chéo với *j*trong{\ displaystyle (i, j)}. Điều này mang lại{\ displaystyle O (n ^ {2})}dung dịch. Điều này có thể được tối ưu hoá hơn nữa khi chúng ta không cần phải xem xét tất cả các dải{\ displaystyle (i, j)} mà thay vào đó chỉ là {\ displaystyle (1, j)}. Thuật toán sau đây do đó tạo ra một{\ displaystyle O (n \ log n)} dung dịch:

1 Weighted - Activity - Selection ( S ) : *// S = danh sách các hoạt động*

2

3 loại S theo thời gian kết thúc 4 chọn [ 0 ] = 0 5 6 **cho** i = 1 đến n : 7 t = tìm kiếm nhị phân để tìm hoạt động với kết thúc thời gian <= thời gian bắt đầu **cho** i 8 lựa chọn

[ I ] = MAX ( opt [ i - 1 ], chọn [ t ] + w ( i ))

9

10 **trở**  opt [ n ]

## Câu 17: Mô tả[[sửa](https://vi.wikipedia.org/w/index.php?title=Thu%E1%BA%ADt_to%C3%A1n_Prim&veaction=edit&section=1) | [sửa mã nguồn](https://vi.wikipedia.org/w/index.php?title=Thu%E1%BA%ADt_to%C3%A1n_Prim&action=edit&section=1)]



Thuật toán Prim có nhiều ứng dụng, chẳng hạn như [xây dựng](https://vi.wikipedia.org/w/index.php?title=X%C3%A2y_d%E1%BB%B1ng_m%C3%AA_cung&action=edit&redlink=1) mê cung trên, bằng cách áp dụng thuật toán Prim cho một [đồ thị lưới](https://vi.wikipedia.org/w/index.php?title=%C4%90%E1%BB%93_th%E1%BB%8B_l%C6%B0%E1%BB%9Bi&action=edit&redlink=1) có trọng số ngẫu nhiên.

Thuật toán xuất phát từ một cây chỉ chứa đúng một đỉnh và mở rộng từng bước một, mỗi bước thêm một cạnh mới vào cây, cho tới khi bao trùm được tất cả các đỉnh của đồ thị.

* Dữ liệu vào: Một đồ thị có trọng số liên thông với tập hợp đỉnh *V* và tập hợp cạnh *E* (trọng số có thể âm). Đồng thời cũng dùng *V* và *E* để ký hiệu số đỉnh và số cạnh của đồ thị.
* Khởi tạo: *V*mới = {*x*}, trong đó *x* là một đỉnh bất kì (đỉnh bắt đầu) trong *V*, *E*mới = {}
* Lặp lại cho tới khi *V*mới = *V*:
  + Chọn cạnh (*u*, *v*) có trọng số nhỏ nhất thỏa mãn *u* thuộc *V*mới và *v* không thuộc *V*mới (nếu có nhiều cạnh như vậy thì chọn một cạnh bất kì trong chúng)
  + Thêm *v* vào *V*mới, và thêm cạnh (*u*, *v*) vào *E*mới
* Dữ liệu ra: *V*mới và *E*mới là tập hợp đỉnh và tập hợp cạnh của một cây bao trùm nhỏ nhất

Độ phức tạp tính toán[[sửa](https://vi.wikipedia.org/w/index.php?title=Thu%E1%BA%ADt_to%C3%A1n_Prim&veaction=edit&section=2) | [sửa mã nguồn](https://vi.wikipedia.org/w/index.php?title=Thu%E1%BA%ADt_to%C3%A1n_Prim&action=edit&section=2)]

|  |  |
| --- | --- |
| **Cấu trúc dữ liệu tìm cạnh có trọng số nhỏ nhất** | **Độ phức tạp thời gian (tổng cộng)** |
| Tìm kiếm trên [ma trận kề](https://vi.wikipedia.org/wiki/Ma_tr%E1%BA%ADn_k%E1%BB%81) | O(V2) |
| [Đống nhị phân](https://vi.wikipedia.org/wiki/%C4%90%E1%BB%91ng_nh%E1%BB%8B_ph%C3%A2n) và [danh sách kề](https://vi.wikipedia.org/wiki/Danh_s%C3%A1ch_k%E1%BB%81) | O((V + E) log V) = O(E log V) |
| [Đống Fibonacci](https://vi.wikipedia.org/w/index.php?title=%C4%90%E1%BB%91ng_Fibonacci&action=edit&redlink=1) và [danh sách kề](https://vi.wikipedia.org/wiki/Danh_s%C3%A1ch_k%E1%BB%81) | O(E + V log V) |

Một cách lập trình đơn giản sử dụng [ma trận kề](https://vi.wikipedia.org/wiki/Ma_tr%E1%BA%ADn_k%E1%BB%81) và tìm kiếm toàn bộ mảng để tìm cạnh có trọng số nhỏ nhất có thời gian chạy [O](https://vi.wikipedia.org/w/index.php?title=K%C3%BD_hi%E1%BB%87u_O_l%E1%BB%9Bn&action=edit&redlink=1)(*V*2). Bằng cách sử dụng cấu trúc dữ liệu [đống nhị phân](https://vi.wikipedia.org/wiki/%C4%90%E1%BB%91ng_nh%E1%BB%8B_ph%C3%A2n) và [danh sách kề](https://vi.wikipedia.org/wiki/Danh_s%C3%A1ch_k%E1%BB%81), có thể giảm thời gian chạy xuống [O](https://vi.wikipedia.org/w/index.php?title=K%C3%BD_hi%E1%BB%87u_O_l%E1%BB%9Bn&action=edit&redlink=1)(*E* log *V*). Bằng cách sử dụng cấu trúc dữ liệu [đống Fibonacci](https://vi.wikipedia.org/w/index.php?title=%C4%90%E1%BB%91ng_Fibonacci&action=edit&redlink=1) phức tạp hơn, có thể giảm thời gian chạy xuống O(*E* + *V*log *V*), nhanh hơn thuật toán trước khi đồ thị có số cạnh *E*=ω(*V*).

Ví dụ[[sửa](https://vi.wikipedia.org/w/index.php?title=Thu%E1%BA%ADt_to%C3%A1n_Prim&veaction=edit&section=3) | [sửa mã nguồn](https://vi.wikipedia.org/w/index.php?title=Thu%E1%BA%ADt_to%C3%A1n_Prim&action=edit&section=3)]

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Hình minh họa** | **U** | **Cạnh (u,v)** | **V \ U** | **Mô tả** |
| [Prim Algorithm 0.svg](https://vi.wikipedia.org/wiki/T%E1%BA%ADp_tin:Prim_Algorithm_0.svg) | {} |  | {A,B,C,D,E,F,G} | Đây là đồ thị có trọng số ban đầu. Các số là các trọng số của các cạnh. |
| [Prim Algorithm 1.svg](https://vi.wikipedia.org/wiki/T%E1%BA%ADp_tin:Prim_Algorithm_1.svg) | {D} | (D,A) = 5 **V** (D,B) = 9 (D,E) = 15 (D,F) = 6 | {A,B,C,E,F,G} | Chọn một cách tùy ý đỉnh **D** là đỉnh bắt đầu. Các đỉnh **A**, **B**, **E** và **F** đều được nối trực tiếp tới **D** bằng cạnh của đồ thị. **A** là đỉnh gần **D** nhất nên ta chọn **A** là đỉnh thứ hai của cây và thêm cạnh **AD** vào cây. |
| [Prim Algorithm 2.svg](https://vi.wikipedia.org/wiki/T%E1%BA%ADp_tin:Prim_Algorithm_2.svg) | {A,D} | (D,B) = 9 (D,E) = 15 (D,F) = 6 **V** (A,B) = 7 | {B,C,E,F,G} | Đỉnh được chọn tiếp theo là đỉnh gần **D** hoặc **A** nhất. **B** có khoảng cách tới **D** bằng 9 và tới **A** bằng 7, **E** có khoảng cách tới cây hiện tại bằng 15, và **F** có khoảng cách bằng 6. **F**là đỉnh gần cây hiện tại nhất nên chọn đỉnh **F** và cạnh **DF**. |
| [Prim Algorithm 3.svg](https://vi.wikipedia.org/wiki/T%E1%BA%ADp_tin:Prim_Algorithm_3.svg) | {A,D,F} | (D,B) = 9 (D,E) = 15 (A,B) = 7 **V** (F,E) = 8 (F,G) = 11 | {B,C,E,G} | Thuật toán tiếp tục tương tự như bước trước. Chọn đỉnh **B** có khoảng cách tới **A** bằng 7. |
| [Prim Algorithm 4.svg](https://vi.wikipedia.org/wiki/T%E1%BA%ADp_tin:Prim_Algorithm_4.svg) | {A,B,D,F} | (B,C) = 8 (B,E) = 7 **V** (D,B) = 9 chu trình (D,E) = 15 (F,E) = 8 (F,G) = 11 | {C,E,G} | Ở bước này ta chọn giữa **C**, **E**, và **G**. **C** có khoảng cách tới **B** bằng 8, **E** có khoảng cách tới **B** bằng 7, và **G** có khoảng cách tới **F** bằng 11. **E** là đỉnh gần nhất, nên chọn đỉnh **E**và cạnh **BE**. |
| [Prim Algorithm 5.svg](https://vi.wikipedia.org/wiki/T%E1%BA%ADp_tin:Prim_Algorithm_5.svg) | {A,B,D,E,F} | (B,C) = 8 (D,B) = 9 chu trình (D,E) = 15 chu trình (E,C) = 5 **V** (E,G) = 9 (F,E) = 8 chu trình (F,G) = 11 | {C,G} | Ở bước này ta chọn giữa **C** và **G**. **C** có khoảng cách tới **E** bằng 5, và **G** có khoảng cách tới **E** bằng 9. Chọn **C** và cạnh **EC**. |
| [Prim Algorithm 6.svg](https://vi.wikipedia.org/wiki/T%E1%BA%ADp_tin:Prim_Algorithm_6.svg) | {A,B,C,D,E,F} | (B,C) = 8 chu trình (D,B) = 9 chu trình (D,E) = 15 chu trình (E,G) = 9 **V** (F,E) = 8 chu trình (F,G) = 11 | {G} | Đỉnh **G** là đỉnh còn lại duy nhất. Nó có khoảng cách tới **F** bằng 11, và khoảng cách tới **E**bằng 9. **E** ở gần hơn nên chọn đỉnh **G** và cạnh **EG**. |
| [Prim Algorithm 7.svg](https://vi.wikipedia.org/wiki/T%E1%BA%ADp_tin:Prim_Algorithm_7.svg) | {A,B,C,D,E,F,G} | (B,C) = 8 chu trình (D,B) = 9 chu trình (D,E) = 15 chu trình (F,E) = 8 chu trình (F,G) = 11 chu trình | {} | Hiện giờ tất cả các đỉnh đã nằm trong cây và [cây bao trùm nhỏ nhất](https://vi.wikipedia.org/wiki/C%C3%A2y_bao_tr%C3%B9m_nh%E1%BB%8F_nh%E1%BA%A5t) được tô màu xanh lá cây. Tổng trọng số của cây là 39. |

## Câu 18: Mô tả thuật toán[[sửa](https://vi.wikipedia.org/w/index.php?title=Thu%E1%BA%ADt_to%C3%A1n_Kruskal&veaction=edit&section=3) | [sửa mã nguồn](https://vi.wikipedia.org/w/index.php?title=Thu%E1%BA%ADt_to%C3%A1n_Kruskal&action=edit&section=3)]

Giả sử ta cần tìm cây bao trùm nhỏ nhất của đồ thị *G*. Thuật toán bao gồm các bước sau.

* Khởi tạo rừng *F* (tập hợp các [cây](https://vi.wikipedia.org/wiki/C%C3%A2y_(l%C3%BD_thuy%E1%BA%BFt_%C4%91%E1%BB%93_th%E1%BB%8B))), trong đó mỗi đỉnh của *G* tạo thành một cây riêng biệt
* Khởi tạo tập *S* chứa tất cả các cạnh của *G*
* Chừng nào *S* còn [khác rỗng](https://vi.wikipedia.org/w/index.php?title=Kh%C3%A1c_r%E1%BB%97ng&action=edit&redlink=1) và *F* gồm hơn một cây
  + Xóa cạnh nhỏ nhất trong *S*
  + Nếu cạnh đó nối hai cây khác nhau trong *F*, thì thêm nó vào *F* và hợp hai cây kề với nó làm một
  + Nếu không thì loại bỏ cạnh đó.

Khi thuật toán kết thúc, rừng chỉ gồm đúng một cây và đó là một cây bao trùm nhỏ nhất của đồ thị *G*.

Mã giả[[sửa](https://vi.wikipedia.org/w/index.php?title=Thu%E1%BA%ADt_to%C3%A1n_Kruskal&veaction=edit&section=4) | [sửa mã nguồn](https://vi.wikipedia.org/w/index.php?title=Thu%E1%BA%ADt_to%C3%A1n_Kruskal&action=edit&section=4)]

Cho đồ thị **G=(X, E)**.

Bước 1: Sắp xếp các cạnh của đồ thị theo thứ tự trọng số tăng dần.

Bước 2: Khởi tạo T:= Ø

Bước 3: Lần lượt lấy từng cạnh thuộc danh sách đã sắp xếp. Nếu T+{e} không chứa chu trình thì gán T:=T+{e}.

Bước 4: Nếu T đủ n-1 phần tử thì dừng, ngược lại làm tiếp bước 2.

**Kỹ thuật đánh nhãn đỉnh**[[sửa](https://vi.wikipedia.org/w/index.php?title=Thu%E1%BA%ADt_to%C3%A1n_Kruskal&veaction=edit&section=5) | [sửa mã nguồn](https://vi.wikipedia.org/w/index.php?title=Thu%E1%BA%ADt_to%C3%A1n_Kruskal&action=edit&section=5)]

Kỹ thuật đánh nhãn đỉnh Trong thuật toán Kruskal, để kiểm tra xem T + {e} có chứa [chu trình](https://vi.wikipedia.org/wiki/Chu_tr%C3%ACnh_(l%C3%BD_thuy%E1%BA%BFt_%C4%91%E1%BB%93_th%E1%BB%8B)) hay không ta có thể dùng kỹ thuật gắn nhãn đỉnh, kỹ thuật này khá đơn giản và hiệu quả.

* Ngay sau **bước 1** của thuật toán, ta gắn đỉnh i của đồ thị một nhãn là i
* **Trong bước 2:**
  + Nếu hai đầu cạnh e có cùng nhãn (tức là nhãn của e.v1 và nhãn của e.v2 bằng nhau) thì T+{e} tạo chu trình, ta không đưa e vào T.
  + Ngược lại [nếu Label(e.v1)!= Label(e.v2) ] thì ta đưa e vào T và thực hiện công việc ghép nhãn bằng cách:
    - lab1 = Min(Label(e.v1), Label (e.v2))
    - lab2 = Max(Label(e.v1), Label (e.v2))
    - Sửa nhãn của tất cả các đỉnh nào có nhãn là lab2 thành nhãn lab1

**Ghi chú**[[sửa](https://vi.wikipedia.org/w/index.php?title=Thu%E1%BA%ADt_to%C3%A1n_Kruskal&veaction=edit&section=6) | [sửa mã nguồn](https://vi.wikipedia.org/w/index.php?title=Thu%E1%BA%ADt_to%C3%A1n_Kruskal&action=edit&section=6)]

* Trong quá trình xây dựng T thì các cạnh có thể không [liên thông](https://vi.wikipedia.org/wiki/T%E1%BA%ADp_h%E1%BB%A3p_li%C3%AAn_th%C3%B4ng) nhau lúc đó T chỉ là [rừng](https://vi.wikipedia.org/wiki/R%E1%BB%ABng) chứ chưa trở thành [cây](https://vi.wikipedia.org/wiki/C%C3%A2y).
* Khi [thuật toán](https://vi.wikipedia.org/wiki/Thu%E1%BA%ADt_to%C3%A1n) dừng:
  + Nếu T chưa đủ n - 1 cạnh thì [đồ thị](https://vi.wikipedia.org/wiki/%C4%90%E1%BB%93_th%E1%BB%8B) G không liên thông(không có cây khung)
  + Ngược lại thì T là [cây khung](https://vi.wikipedia.org/wiki/C%C3%A2y_bao_tr%C3%B9m) cần tìm.

Thời gian thực hiện[[sửa](https://vi.wikipedia.org/w/index.php?title=Thu%E1%BA%ADt_to%C3%A1n_Kruskal&veaction=edit&section=7) | [sửa mã nguồn](https://vi.wikipedia.org/w/index.php?title=Thu%E1%BA%ADt_to%C3%A1n_Kruskal&action=edit&section=7)]

* Nếu *E* là số cạnh và *V* là số đỉnh của đồ thị thì thuật toán Kruskal chạy trong thời gian [*O*](https://vi.wikipedia.org/wiki/K%C3%AD_hi%E1%BB%87u_O_l%E1%BB%9Bn)(*E* [log](https://vi.wikipedia.org/wiki/L%C3%B4garit) *V*).
* Có thể đạt được thời gian này bằng phương pháp sau: [sắp xếp](https://vi.wikipedia.org/wiki/Thu%E1%BA%ADt_to%C3%A1n_s%E1%BA%AFp_x%E1%BA%BFp) tất cả các cạnh theo trọng số trong thời gian *O*(*E* log *E*). Điều này cho phép thực hiện bước "xóa cạnh nhỏ nhất trong *S*" trong thời gian hằng số. Sau đó sử dụng [cấu trúc dữ liệu cho các tập hợp không giao nhau](https://vi.wikipedia.org/wiki/C%E1%BA%A5u_tr%C3%BAc_d%E1%BB%AF_li%E1%BB%87u_cho_c%C3%A1c_t%E1%BA%ADp_h%E1%BB%A3p_kh%C3%B4ng_giao_nhau) để lưu trữ thông tin đỉnh nào nằm ở cây nào trong *F*. Ta cần thực hiện O(*E*) thao tác, hai thao tác 'tìm' và không quá một thao tác 'hợp' cho mỗi cạnh. Ngay cả những thuật toán đơn giản cho bài toán này, chẳng hạn hợp bằng trọng số cũng có thể thực hiện O(*E*) thao tác trong thời gian *O*(*E* log *V*). Vì vậy tổng thời gian là *O*(*E* log *E*) = *O*(*E* log *V*).

Chứng minh tính đúng đắn[[sửa](https://vi.wikipedia.org/w/index.php?title=Thu%E1%BA%ADt_to%C3%A1n_Kruskal&veaction=edit&section=8) | [sửa mã nguồn](https://vi.wikipedia.org/w/index.php?title=Thu%E1%BA%ADt_to%C3%A1n_Kruskal&action=edit&section=8)]

Chứng minh gồm hai phần: chứng minh kết quả thuật toán là một cây bao trùm và cây bao trùm đó là nhỏ nhất.

[**Cây bao trùm**](https://vi.wikipedia.org/wiki/C%C3%A2y_bao_tr%C3%B9m)[[sửa](https://vi.wikipedia.org/w/index.php?title=Thu%E1%BA%ADt_to%C3%A1n_Kruskal&veaction=edit&section=9) | [sửa mã nguồn](https://vi.wikipedia.org/w/index.php?title=Thu%E1%BA%ADt_to%C3%A1n_Kruskal&action=edit&section=9)]

*F* luôn là một rừng do việc nối hai cây bằng một cạnh luôn tạo ra một cây mới. Giả thiết phản chứng *F* gồm ít nhất hai cây *A* và *B*. Khi cạnh đầu tiên nối các đỉnh trong *A* của *F* với phần còn lại của đồ thị được xem xét (cạnh này tồn tại do *G* liên thông) thì rõ ràng thuật toán sẽ chọn nó. Vì vậy *A* không thể là một cây trong *F* khi thuật toán kết thúc. Do đó, *F* liên thông và là một cây bao trùm.

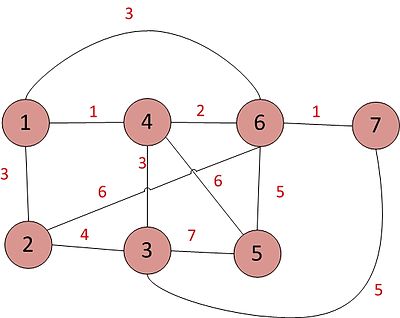
**Nhỏ nhất**[[sửa](https://vi.wikipedia.org/w/index.php?title=Thu%E1%BA%ADt_to%C3%A1n_Kruskal&veaction=edit&section=10) | [sửa mã nguồn](https://vi.wikipedia.org/w/index.php?title=Thu%E1%BA%ADt_to%C3%A1n_Kruskal&action=edit&section=10)]

Ta chứng minh mệnh đề ***P*** sau đây bằng [quy nạp](https://vi.wikipedia.org/wiki/Quy_n%E1%BA%A1p_to%C3%A1n_h%E1%BB%8Dc): Nếu *F* là tập hợp các cạnh đã chọn tại bất kì thời điểm nào trong quá trình thực thi thuật toán thì tồn tại cây bao trùm nhỏ nhất chứa *F*.

* Rõ ràng ***P*** đúng khi thuật toán bắt đầu vì *F* là rỗng.
* Giả sử ***P*** là đúng cho một tập hợp *F* và giả sử *T* là một cây bao trùm nhỏ nhất chứa *F*. Nếu cạnh được thêm vào tiếp theo là *e* cũng nằm trong *T*, thì ***P*** đúng cho *F* + *e*. Nếu không, thì *T* + *e* chứa chu trình *C* và tồn tại cạnh *f* nằm trên *C* nhưng không trong *F*. (Nếu không có cạnh *f*, thì không thể thêm *e* vào *F*, do sẽ tạo ra chu trình *C* trong *F*.) Do đó *T*− *f* + *e* là một cây, và nó có cùng trọng số với *T*, do *T* có trọng số nhỏ nhất và *f* không thể nhỏ hơn *e*, vì nếu không thuật toán đã xem xét *f* trước *e* và chọn *f*. Vì vậy *T* − *f* + *e* là một cây bao trùm nhỏ nhất chứa *F* + *e* và ***P*** là đúng.
* Như vậy, ***P*** đúng khi thuật toán kết thúc và *F* là một cây bao trùm. Điều này chỉ có thể xảy ra nếu *F* là một cây bao trùm nhỏ nhất.

Ví dụ[[sửa](https://vi.wikipedia.org/w/index.php?title=Thu%E1%BA%ADt_to%C3%A1n_Kruskal&veaction=edit&section=11) | [sửa mã nguồn](https://vi.wikipedia.org/w/index.php?title=Thu%E1%BA%ADt_to%C3%A1n_Kruskal&action=edit&section=11)]

* **Cho đồ thị G như hình vẽ:**. Yêu cầu tìm ra cây khung nhỏ nhất của đồ thị G.
* **G gồm có 7 đỉnh**
* Đồ thị G có n phần tử. Thuật toán Kruskal sẽ dừng khi có n-1 trong tập hợp T
  + n = 7
  + Vậy số cạnh trong tập hợp T: n - 1 = 7 - 1 = 6(\*)

[](https://vi.wikipedia.org/wiki/T%E1%BA%ADp_tin:%C4%90%E1%BB%93_th%E1%BB%8B_G_trong_l%C3%BD_thuy%E1%BA%BFt_%C4%91%E1%BB%93_th%E1%BB%8B.jpg)

Đồ thị G

**Bước 1: Liệt kê tất cả cạnh với trọng số của cạnh đó:** Dựa vào đồ thị ta liệt kê ra các cạnh gồm đỉnh đầu, đỉnh cuối và trọng số:

|  |  |  |
| --- | --- | --- |
| **Điểm đầu** | **Điểm cuối** | **Trọng số** |
| 1 | 2 | 3 |
| 1 | 4 | 1 |
| 1 | 6 | 3 |
| 2 | 3 | 4 |
| 2 | 6 | 6 |
| 3 | 4 | 3 |
| 3 | 5 | 7 |
| 3 | 7 | 5 |
| 4 | 5 | 6 |
| 4 | 6 | 2 |
| 5 | 6 | 5 |
| 6 | 7 | 1 |

**Bước 2: Sắp xếp các cạnh theo trọng số tăng dần:**

|  |  |  |
| --- | --- | --- |
| **Điểm đầu** | **Điểm cuối** | **Trọng số** |
| 1 | 4 | 1 |
| 6 | 7 | 1 |
| 4 | 6 | 2 |
| 1 | 2 | 3 |
| 1 | 6 | 3 |
| 3 | 4 | 3 |
| 2 | 3 | 4 |
| 3 | 7 | 5 |
| 5 | 6 | 5 |
| 2 | 6 | 6 |
| 4 | 5 | 6 |
| 3 | 5 | 7 |

**Bước 3: Dựa vào kết quả ở bước 2. Ta tiến hành tìm cây khung bằng thuật toán Kruskal**

|  |  |
| --- | --- |
| **Kết quả** | **Cạnh đang xét** |
| [https://upload.wikimedia.org/wikipedia/commons/thumb/4/41/Do_thi_G_tung_buoc_%281%29.png/200px-Do_thi_G_tung_buoc_%281%29.png](https://vi.wikipedia.org/wiki/T%E1%BA%ADp_tin:Do_thi_G_tung_buoc_(1).png)  Đồ thị G | 1-4-1: Ta nhận thấy cạnh 1-4 không tạo ra một chu trình nào. Vì vậy, thêm 1-4 vào tập hợp |
| [https://upload.wikimedia.org/wikipedia/commons/thumb/b/b0/Do_thi_G_tung_buoc_%282%29.png/200px-Do_thi_G_tung_buoc_%282%29.png](https://vi.wikipedia.org/wiki/T%E1%BA%ADp_tin:Do_thi_G_tung_buoc_(2).png)  Đồ thị G | 6-7-1: Ta nhận thấy cạnh 6-7 không tạo ra một chu trình nào. Vì vậy, thêm 6-7 vào tập hợp |
| [https://upload.wikimedia.org/wikipedia/commons/thumb/6/6f/Do_thi_G_tung_buoc_%283%29.png/200px-Do_thi_G_tung_buoc_%283%29.png](https://vi.wikipedia.org/wiki/T%E1%BA%ADp_tin:Do_thi_G_tung_buoc_(3).png)  Đồ thị G | 4-6-2: Ta nhận thấy cạnh 4-6 không tạo ra một chu trình nào. Vì vậy, thêm 4-6 vào tập hợp |
| [https://upload.wikimedia.org/wikipedia/commons/thumb/c/c6/Do_thi_G_tung_buoc_%284%29.png/200px-Do_thi_G_tung_buoc_%284%29.png](https://vi.wikipedia.org/wiki/T%E1%BA%ADp_tin:Do_thi_G_tung_buoc_(4).png)  Đồ thị G | 1-2-3: Ta nhận thấy cạnh 1-2 không tạo ra một chu trình nào. Vì vậy, thêm 1-2 vào tập hợp |
| [https://upload.wikimedia.org/wikipedia/commons/thumb/b/b7/Do_thi_G_tung_buoc_%285%29.png/200px-Do_thi_G_tung_buoc_%285%29.png](https://vi.wikipedia.org/wiki/T%E1%BA%ADp_tin:Do_thi_G_tung_buoc_(5).png)  Đồ thị G | 1-6-3: Ta nhận thấy cạnh 1-6 tạo ra một chu trình. Không thêm vào tập hợp.  3-4-3: Ta nhận thấy cạnh 3-4 không tạo ra một chu trình. Vì vậy, thêm 3-4 vào tập hợp |
| [https://upload.wikimedia.org/wikipedia/commons/thumb/0/04/Do_thi_G_tung_buoc_%286%29.png/200px-Do_thi_G_tung_buoc_%286%29.png](https://vi.wikipedia.org/wiki/T%E1%BA%ADp_tin:Do_thi_G_tung_buoc_(6).png)  Đồ thị G | 2-3-4: Ta nhận thấy cạnh 2-3 tạo ra một chu trình. Không thêm vào tập hợp. |
| [https://upload.wikimedia.org/wikipedia/commons/thumb/3/31/Do_thi_G_tung_buoc_%287%29.png/200px-Do_thi_G_tung_buoc_%287%29.png](https://vi.wikipedia.org/wiki/T%E1%BA%ADp_tin:Do_thi_G_tung_buoc_(7).png)  Đồ thị G | 3-7-5: Ta nhận thấy cạnh 3-7 tạo ra một chu trình. Không thêm vào tập hợp. |
| [https://upload.wikimedia.org/wikipedia/commons/thumb/5/5b/Do_thi_G_tung_buoc_%288%29.png/200px-Do_thi_G_tung_buoc_%288%29.png](https://vi.wikipedia.org/wiki/T%E1%BA%ADp_tin:Do_thi_G_tung_buoc_(8).png)  Đồ thị G | 5-6-5: Ta nhận thấy cạnh 5-6 không tạo ra một chu trình nào. Vì vậy, thêm 5-6 vào tập hợp |

* Đến đây, ta đã tìm được 6 cạnh. Vậy kết thúc thuật toán. (Thỏa (\*))
* **Kết quả: Ta được đồ thị sau**

|  |
| --- |
|  |
| [https://upload.wikimedia.org/wikipedia/commons/thumb/7/7b/Do_thi_G_tung_buoc_%289%29.png/400px-Do_thi_G_tung_buoc_%289%29.png](https://vi.wikipedia.org/wiki/T%E1%BA%ADp_tin:Do_thi_G_tung_buoc_(9).png)  Đồ thị G |

**Với tổng chi phí là: Ta cộng tất cả các trọng số giữa các đỉnh lại với nhau**

* Vậy tổng chi phí: 3 + 1 + 3 + 2 + 5 + 1 = 15

## Câu 19: Tư tưởng thuật toán[[sửa](https://vi.wikipedia.org/w/index.php?title=Thu%E1%BA%ADt_to%C3%A1n_Bellman-Ford&veaction=edit&section=1) | [sửa mã nguồn](https://vi.wikipedia.org/w/index.php?title=Thu%E1%BA%ADt_to%C3%A1n_Bellman-Ford&action=edit&section=1)]

[[1]](https://vi.wikipedia.org/wiki/Thu%E1%BA%ADt_to%C3%A1n_Bellman-Ford#cite_note-1)

- Bước 1: Khởi tạo ∏(0,x)=0; ∏(0,i)=+∞, ∀i≠x và k=1

- Bước 2: Với mỗi i∈X ta đặt:

∏(k,i)=min({∏(k-1,i)}∪{∏(k-1,j)+L[j][i]})

- Bước 3: Nếu ∏(k,i)=∏(k-1,i) với i∈X thì ∏(k,i) chính là độ dài đường đi ngắn nhất từ x đến i. Ngược lại nếu k<n thì tăng k=k+1 và trở lại bước 2; nếu k=n thì dừng vì từ x đi tới được 1 mạch âm.

**Ưu điểm:**[[2]](https://vi.wikipedia.org/wiki/Thu%E1%BA%ADt_to%C3%A1n_Bellman-Ford#cite_note-2)

- Từ 1 đỉnh xuất phát nhìn hình ta có thế suy ra đường đi ngắn nhất từ đỉnh đó tới các đỉnh khác mà không cần làm lại từ đầu.

- Ví dụ: Từ đỉnh 1 ta có thể tìm đường đi ngắn nhất từ 1->3 và 1->4 mà không cần làm lại.

Nội dung thuật toán[[sửa](https://vi.wikipedia.org/w/index.php?title=Thu%E1%BA%ADt_to%C3%A1n_Bellman-Ford&veaction=edit&section=2) | [sửa mã nguồn](https://vi.wikipedia.org/w/index.php?title=Thu%E1%BA%ADt_to%C3%A1n_Bellman-Ford&action=edit&section=2)]

**function** BellmanFord(danh\_sách\_đỉnh, danh\_sách\_cung, nguồn)

*// hàm yêu cầu đồ thị đưa vào dưới dạng một danh sách đỉnh, một danh sách cung*

*// hàm tính các giá trị* khoảng\_cách *và* đỉnh\_liền\_trước *của các đỉnh,*

*// sao cho các giá trị* đỉnh\_liền\_trước *sẽ lưu lại các đường đi ngắn nhất.*

*// bước 1: khởi tạo đồ thị*

**for each** v **in** danh\_sách\_đỉnh:

**if** v **is** nguồn **then** khoảng\_cách(v):= 0

**else** khoảng\_cách(v):= **vô cùng**

đỉnh\_liền\_trước(v):= **null**

*// bước 2: kết nạp cạnh*

**for** i **from** 1 **to** size(danh\_sách\_đỉnh)-1:

**for each** (u,v) **in** danh\_sách\_cung:

**if** khoảng\_cách(v) > khoảng\_cách(u) + trọng\_số(u,v):

khoảng\_cách(v):= khoảng\_cách(u) + trọng\_số(u,v)

đỉnh\_liền\_trước(v):= u

*// bước 3: kiểm tra chu trình âm*

**for each** (u,v) **in** danh\_sách\_cung:

**if** khoảng\_cách(v) > khoảng\_cách(u) + trọng\_số(u,v):

**error** "Đồ thị chứa chu trình âm"

Chứng minh tính đúng đắn[[sửa](https://vi.wikipedia.org/w/index.php?title=Thu%E1%BA%ADt_to%C3%A1n_Bellman-Ford&veaction=edit&section=3) | [sửa mã nguồn](https://vi.wikipedia.org/w/index.php?title=Thu%E1%BA%ADt_to%C3%A1n_Bellman-Ford&action=edit&section=3)]

Tính đúng đắn của thuật toán có thể được chứng minh bằng [quy nạp](https://vi.wikipedia.org/wiki/Quy_n%E1%BA%A1p_to%C3%A1n_h%E1%BB%8Dc). Thuật toán có thể được phát biểu chính xác theo kiểu quy nạp như sau:

**Bổ đề**. Sau *i* lần lặp vòng *for*:

1. Nếu Khoảng\_cách(u) không có giá trị vô cùng lớn, thì nó bằng độ dài của một đường đi nào đó từ *s* tới *u*;
2. Nếu có một đường đi từ *s* tới *u* qua nhiều nhất *i* cung, thì Khoảng\_cách(u) có giá trị không vượt quá độ dài của đường đi ngắn nhất từ *s* tới *u* qua tối đa *i* cung.

**Chứng minh**.

Trường hợp cơ bản: Xét i=0 và thời điểm trước khi vòng *for* được chạy lần đầu tiên. Khi đó, với đỉnh nguồn khoảng\_cách(nguồn) = 0, điều này đúng. Đối với các đỉnh *u* khác, khoảng\_cách(u) = **vô cùng**, điều này cũng đúng vì không có đường đi nào từ *nguồn* đến *u* qua 0 cung.

Trường hợp quy nạp:

Chứng minh câu 1. Xét thời điểm khi khoảng cách tới một đỉnh được cập nhật bởi công thức khoảng\_cách(v):= khoảng\_cách(u) + trọng\_số(u,v). Theo giả thiết quy nạp, khoảng\_cách(u) là độ dài của một đường đi nào đó từ *nguồn* tới *u*. Do đó, khoảng\_cách(u) + trọng\_số(u,v) là độ dài của đường đi từ *nguồn* tới *u* rồi tới *v*.

Chứng minh câu 2: Xét đường đi ngắn nhất từ *nguồn* tới *u* qua tối đa *i* cung. Giả sử *v* là đỉnh liền ngay trước *u* trên đường đi này. Khi đó, phần đường đi từ *nguồn* tới *v* là đường đi ngắn nhất từ *nguồn* tới *v* qua tối đa *i-1* cung. Theo giả thuyết quy nạp, khoảng\_cách(v) sau *i-1* vòng lặp không vượt quá độ dài đường đi này. Do đó, trọng\_số(v,u) + khoảng\_cách(v) có giá trị không vượt quá độ dài của đường đi từ *s* tới *u*. Trong lần lặp thứ *i*, khoảng\_cách(u) được lấy giá trị nhỏ nhất của khoảng\_cách(v) + trọng\_số(v,u) với mọi *v* có thể. Do đó, sau *i* lần lặp, khoảng\_cách(u) có giá trị không vượt quá độ dài đường đi ngắn nhất từ *nguồn* tới *u* qua tối đa *i* cung.

Khi *i* bằng số đỉnh của đồ thị, mỗi đường đi tìm được sẽ là đường đi ngắn nhất toàn cục, trừ khi đồ thị có chu trình âm. Nếu tồn tại chu trình âm mà từ đỉnh nguồn có thể đi đến được thì sẽ không tồn tại đường đi nhỏ nhất (vì mỗi lần đi quanh chu trình âm là một lần giảm trọng số của đường).

Ứng dụng trong định tuyến[[sửa](https://vi.wikipedia.org/w/index.php?title=Thu%E1%BA%ADt_to%C3%A1n_Bellman-Ford&veaction=edit&section=4) | [sửa mã nguồn](https://vi.wikipedia.org/w/index.php?title=Thu%E1%BA%ADt_to%C3%A1n_Bellman-Ford&action=edit&section=4)]

Một biến thể phân tán của thuật toán Bellman-Ford được dùng trong các [giao thức định tuyến vector khoảng cách](https://vi.wikipedia.org/w/index.php?title=Giao_th%E1%BB%A9c_%C4%91%E1%BB%8Bnh_tuy%E1%BA%BFn_vector_kho%E1%BA%A3ng_c%C3%A1ch&action=edit&redlink=1), chẳng hạn [giao thức RIP](https://vi.wikipedia.org/w/index.php?title=Giao_th%E1%BB%A9c_RIP&action=edit&redlink=1) (*Routing Information Protocol*). Đây là biến thể phân tán vì nó liên quan đến các nút mạng (các [thiết bị định tuyến](https://vi.wikipedia.org/wiki/Router)) trong một [hệ thống tự chủ](https://vi.wikipedia.org/w/index.php?title=H%E1%BB%87_th%E1%BB%91ng_t%E1%BB%B1_ch%E1%BB%A7_(li%C3%AAn_m%E1%BA%A1ng)&action=edit&redlink=1) (*autonomous system*), ví dụ một tập các mạng IP thuộc sở hữu của một [nhà cung cấp dịch vụ Internet](https://vi.wikipedia.org/wiki/Nh%C3%A0_cung_c%E1%BA%A5p_d%E1%BB%8Bch_v%E1%BB%A5_Internet) (ISP).

Thuật toán gồm các bước sau:

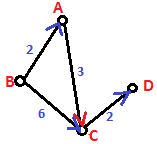
1. Mỗi nút tính khoảng cách giữa nó và tất cả các nút khác trong hệ thống tự chủ và lưu trữ thông tin này trong một bảng.
2. Mỗi nút gửi bảng thông tin của mình cho tất cả các nút lân cận.
3. Khi một nút nhận được các bảng thông tin từ các nút lân cận, nó tính các tuyến đường ngắn nhất tới tất cả các nút khác và cập nhật bảng thông tin của chính mình.

Nhược điểm chính của thuật toán Bellman-Ford trong cấu hình này là

* Không nhân rộng tốt
* Các thay đổi của [tô-pô mạng](https://vi.wikipedia.org/w/index.php?title=T%C3%B4-p%C3%B4_m%E1%BA%A1ng&action=edit&redlink=1) không được ghi nhận nhanh do các cập nhật được lan truyền theo từng nút một.
* Đếm dần đến [vô cùng](https://vi.wikipedia.org/wiki/V%C3%B4_t%E1%BA%ADn) (nếu liên kết hỏng hoặc nút mạng hỏng làm cho một nút bị tách khỏi một tập các nút khác, các nút này vẫn sẽ tiếp tục ước tính khoảng cách tới nút đó và tăng dần giá trị tính được, trong khi đó còn có thể xảy ra việc định tuyến thành vòng tròn)

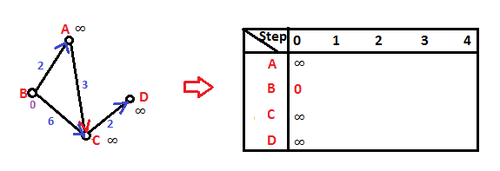
Minh họa bằng hình[[sửa](https://vi.wikipedia.org/w/index.php?title=Thu%E1%BA%ADt_to%C3%A1n_Bellman-Ford&veaction=edit&section=5) | [sửa mã nguồn](https://vi.wikipedia.org/w/index.php?title=Thu%E1%BA%ADt_to%C3%A1n_Bellman-Ford&action=edit&section=5)]

**Tìm đường đi ngắn nhất từ đỉnh B tới đỉnh D của đồ thị G** [[3]](https://vi.wikipedia.org/wiki/Thu%E1%BA%ADt_to%C3%A1n_Bellman-Ford#cite_note-3)

[](https://vi.wikipedia.org/wiki/T%E1%BA%ADp_tin:DothiGcotrongso.PNG)

Đồ thị G

- **Bước 0:** Ta đánh dấu đỉnh xuất phát = 0, các đinh còn lại bằng vô cực.

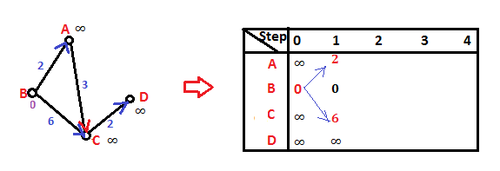
[](https://vi.wikipedia.org/wiki/T%E1%BA%ADp_tin:Buoc0.png)

Bước 0

- **Bước 1:**

Tại đỉnh A có đỉnh B đi vào có chi phí hiện tại (2) < chi phí trước (∞) => cập nhật lại chi phí đỉnh A

Tại đỉnh C có đỉnh B đi vào có chi phí hiện tại (6) < chi phí trước (∞) => cập nhật lại chi phí đỉnh C

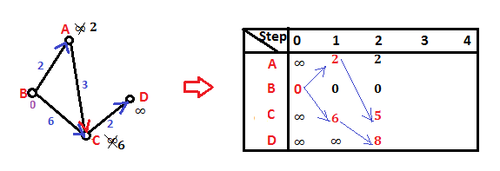
[](https://vi.wikipedia.org/wiki/T%E1%BA%ADp_tin:Buoc1.png)

Bước 1

- **Bước 2:**

Tại đỉnh C có đỉnh A đi vào có chi phí hiện tại (5) < chi phí trước (6) => cập nhật lại chi phí đỉnh C

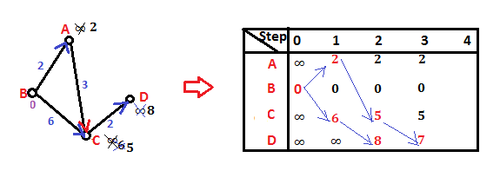
Tại đỉnh D có đỉnh C đi vào có chi phí hiện tại (8) < chi phí trước (∞) => cập nhật lại chi phí đỉnh D

[](https://vi.wikipedia.org/wiki/T%E1%BA%ADp_tin:Buoc2.png)

Bước 2

- **Bước 3:**

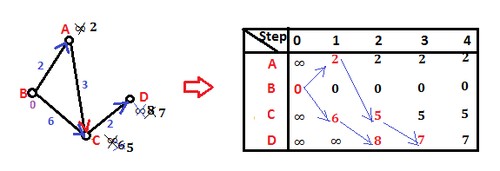
Tại đỉnh D có đỉnh C đi vào có chi phí hiện tại (7) < chi phí trước (8) => cập nhật lại chi phí đỉnh D

[](https://vi.wikipedia.org/wiki/T%E1%BA%ADp_tin:Buoc3.png)

Bước 3

- **Bước 4:**

Bước 4 giống bước 3 nên thuật toán dừng.

[](https://vi.wikipedia.org/wiki/T%E1%BA%ADp_tin:Buoc4.png)

Bước 4

- **Kết luận:**

Có đường đi ngắn nhất từ B->D: B->A->C->D

- **Lưu ý:**

- Nếu Bước 4 không giống bước 3 => kết luận không có đường đi ngắn nhất từ B->D

Câu 20: **huật toán Floyd-Warshall** còn được gọi là thuật toán Floyd được [Robert Floyd](https://en.wikipedia.org/wiki/Robert_Floyd) tìm ra năm 1962.thuật toán Floyd là một [thuật toán](https://vi.wikipedia.org/wiki/Thu%E1%BA%ADt_to%C3%A1n) giải quyết [bài toán đường đi ngắn nhất](https://vi.wikipedia.org/wiki/B%C3%A0i_to%C3%A1n_%C4%91%C6%B0%E1%BB%9Dng_%C4%91i_ng%E1%BA%AFn_nh%E1%BA%A5t) trong một [đồ thị có hướng](https://vi.wikipedia.org/wiki/%C4%90%E1%BB%93_th%E1%BB%8B_c%C3%B3_h%C6%B0%E1%BB%9Bng) có [cạnh](https://vi.wikipedia.org/w/index.php?title=C%E1%BA%A1nh&action=edit&redlink=1) mang trọng số [**dương**](https://vi.wikipedia.org/wiki/S%E1%BB%91_d%C6%B0%C6%A1ng) dựa trên khái niệm **các**[**Đỉnh Trung Gian**](https://vi.wikipedia.org/w/index.php?title=%C4%90%E1%BB%89nh_Trung_Gian&action=edit&redlink=1).

**Bài toán:** Xét [đồ thị có hướng](https://vi.wikipedia.org/wiki/%C4%90%E1%BB%93_th%E1%BB%8B_c%C3%B3_h%C6%B0%E1%BB%9Bng) có trọng số G=<V,E>:

Tập đỉnh: V={v1, v2, …, vn}  
Ma trận khoảng cách: W = (i, j)

Thuật toán Floyd-Warshall giúp xác định tất cả các [Đường đi ngắn nhất](https://vi.wikipedia.org/wiki/%C4%90%C6%B0%E1%BB%9Dng_%C4%91i_ng%E1%BA%AFn_nh%E1%BA%A5t) giữa tất cả các cặp đỉnh.

**Định lý:** Thuật toán Floyd-Warshall cho ta [Ma trận](https://vi.wikipedia.org/wiki/Ma_tr%E1%BA%ADn) W\* = Wn là ma trận [Khoảng cách](https://vi.wikipedia.org/wiki/Kho%E1%BA%A3ng_c%C3%A1ch) nhỏ nhất của [đồ thị](https://vi.wikipedia.org/wiki/%C4%90%E1%BB%93_th%E1%BB%8B) G.

## Chú thích[[sửa](https://vi.wikipedia.org/w/index.php?title=Thu%E1%BA%ADt_to%C3%A1n_Floyd-Warshall&veaction=edit&section=1) | [sửa mã nguồn](https://vi.wikipedia.org/w/index.php?title=Thu%E1%BA%ADt_to%C3%A1n_Floyd-Warshall&action=edit&section=1)]

Có thể hiểu một cách đơn giản. Để đi từ a --> b. Bạn mất 1 quãng đường là x.  
Thuật toán sẽ tìm 1 đường đi gián tiếp từ a -- k -- b và nếu đường đi này ngắn hơn đường đi trực tiếp thì ta gán luôn giá trị nhỏ nhất của đường đi trực tiếp bằng đường đi gián tiếp.  
Thuật toán Floyd cần {\displaystyle O(n^{3})} để giải [Bài toán đường đi ngắn nhất](https://vi.wikipedia.org/wiki/B%C3%A0i_to%C3%A1n_%C4%91%C6%B0%E1%BB%9Dng_%C4%91i_ng%E1%BA%AFn_nh%E1%BA%A5t) cho **mỗi cặp đỉnh.**

## Câu 22: Nội dung[[sửa](https://vi.wikipedia.org/w/index.php?title=B%C3%A0i_to%C3%A1n_x%E1%BA%BFp_ba_l%C3%B4&veaction=edit&section=1) | [sửa mã nguồn](https://vi.wikipedia.org/w/index.php?title=B%C3%A0i_to%C3%A1n_x%E1%BA%BFp_ba_l%C3%B4&action=edit&section=1)]

Một kẻ trộm đột nhập vào một cửa hiệu tìm thấy có *n* mặt hàng có trọng lượng và giá trị khác nhau, nhưng hắn chỉ mang theo một cái túi có sức chứa về trọng lượng tối đa là *M*. Vậy kẻ trộm nên bỏ vào ba lô những món nào và số lượng bao nhiêu để đạt giá trị cao nhất trong khả năng mà hắn có thể mang đi được.

Dạng [bài toán quyết định](https://vi.wikipedia.org/w/index.php?title=B%C3%A0i_to%C3%A1n_quy%E1%BA%BFt_%C4%91%E1%BB%8Bnh&action=edit&redlink=1) của bài toán xếp ba lô là câu hỏi "có thể đạt được một giá trị ít nhất là bao nhiêu theo phát biểu của bài toán".

Ta có *n* loại đồ vật, *x*1 tới *xn*. Mỗi đồ vật *xj* có một giá trị *pj* và một khối lượng *wj*. Khối lượng tối đa mà ta có thể mang trong ba lô là *C*.

**Bài xếp ba lô dạng 0-1**[[sửa](https://vi.wikipedia.org/w/index.php?title=B%C3%A0i_to%C3%A1n_x%E1%BA%BFp_ba_l%C3%B4&veaction=edit&section=2) | [sửa mã nguồn](https://vi.wikipedia.org/w/index.php?title=B%C3%A0i_to%C3%A1n_x%E1%BA%BFp_ba_l%C3%B4&action=edit&section=2)]

Hạn chế số đồ vật thuộc mỗi loại là 0 (không được chọn) và 1 (được chọn).

Bài xếp ba lô 0-1 có thể được phát biểu bằng toán học như sau:

Cực đại hóa {\displaystyle \sum \_{j=1}^{n}p\_{j}x\_{j}.}

sao cho {\displaystyle \sum \_{j=1}^{n}w\_{j}x\_{j}\leq c,\quad \quad x\_{j}=0\;{\mbox{or}}\;1,\quad j=1,\dots ,n.}

**Bài xếp ba lô bị chặn** hạn chế số đồ vật không được vượt quá một lượng nào đó.

Bài xếp ba lô bị chặn có thể được phát biểu bằng toán học như sau:

Cực đại hóa {\displaystyle \sum \_{j=1}^{n}p\_{j}x\_{j}.}

sao cho {\displaystyle \sum \_{j=1}^{n}w\_{j}x\_{j}\leq c,\quad \quad 0\leq x\_{j}\leq b\_{j},\quad j=1,\dots ,n.}

**Bài xếp ba lô không bị chặn** không có một hạn chế nào về số lượng đồ vật.

**Trường hợp đặc biệt**

Bài toán với các tính chất:

* là một bài toán quyết định
* là một bài toán 0/1
* với mỗi đồ vật, chi phí bằng giá trị: *C* = *V*

Lưu ý rằng trong trường hợp đặc biệt này, bài toán tương đương với:

Cho một tập các số nguyên, tồn tại hay không một tập con có tổng đúng bằng *C*?

Hoặc nếu đồ vật được phép có chi phí âm và *C* được chọn bằng 0, bài toán có dạng:

Cho trước một tập các số nguyên, tồn tại hay không một tập con có tổng đúng bằng *0*?

Trường hợp đặc biệt này được gọi là [bài toán tổng các tập con](https://vi.wikipedia.org/w/index.php?title=B%C3%A0i_to%C3%A1n_t%E1%BB%95ng_c%C3%A1c_t%E1%BA%ADp_con&action=edit&redlink=1) (*subset sum problem*). Với một số lý do, trong ngành [mật mã học](https://vi.wikipedia.org/wiki/M%E1%BA%ADt_m%C3%A3_h%E1%BB%8Dc), người ta thường dùng cụm từ "bài toán xếp ba lô" khi thực ra đang có ý nói về "bài toán tổng con".

Bài toán xếp ba lô thường được giải bằng [quy hoạch động](https://vi.wikipedia.org/wiki/Quy_ho%E1%BA%A1ch_%C4%91%E1%BB%99ng), tuy chưa có một [thuật toán thời gian đa thức](https://vi.wikipedia.org/w/index.php?title=Thu%E1%BA%ADt_to%C3%A1n_th%E1%BB%9Di_gian_%C4%91a_th%E1%BB%A9c&action=edit&redlink=1) cho bài toán tổng quát. Cả bài xếp ba lô tổng quát và bài toán tổng con đều là các bài [NP-khó](https://vi.wikipedia.org/wiki/NP-kh%C3%B3), và điều này dẫn đến các cố gắng sử dụng tổng con làm cơ sở cho các hệ thống [mật mã hóa khóa công khai](https://vi.wikipedia.org/wiki/M%E1%BA%ADt_m%C3%A3_h%C3%B3a_kh%C3%B3a_c%C3%B4ng_khai), chẳng hạn [Merkle-Hellman](https://vi.wikipedia.org/w/index.php?title=Merkle-Hellman&action=edit&redlink=1). Các cố gắng này thường dùng [nhóm](https://vi.wikipedia.org/wiki/Nh%C3%B3m_(%C4%91%E1%BA%A1i_s%E1%BB%91)) thay vì các số nguyên. Merkle-Hellman và một số thuật toán tương tự khác đã bị phá, do các bài toán tổng con cụ thể mà họ tạo ra thực ra lại giải được bằng các thuật toán thời gian đa thức.

Phiên bản bài toán quyết định của bài xếp ba lô được mô tả ở trên là [NP-đầy đủ](https://vi.wikipedia.org/wiki/NP-%C4%91%E1%BA%A7y_%C4%91%E1%BB%A7) và trong thực tế là một trong [21 bài toán NP-đầy đủ của Karp](https://vi.wikipedia.org/w/index.php?title=21_b%C3%A0i_to%C3%A1n_NP-%C4%91%E1%BA%A7y_%C4%91%E1%BB%A7_c%E1%BB%A7a_Karp&action=edit&redlink=1).

**Bài xếp ba lô dạng phân số**[[sửa](https://vi.wikipedia.org/w/index.php?title=B%C3%A0i_to%C3%A1n_x%E1%BA%BFp_ba_l%C3%B4&veaction=edit&section=3) | [sửa mã nguồn](https://vi.wikipedia.org/w/index.php?title=B%C3%A0i_to%C3%A1n_x%E1%BA%BFp_ba_l%C3%B4&action=edit&section=3)]

Với mỗi loại, có thể chọn một phần của nó (ví dụ: 1Kg bơ có thể được cắt ra thành nhiều phần để bỏ vào ba lô)

Cách giải bằng quy hoạch động[[sửa](https://vi.wikipedia.org/w/index.php?title=B%C3%A0i_to%C3%A1n_x%E1%BA%BFp_ba_l%C3%B4&veaction=edit&section=4) | [sửa mã nguồn](https://vi.wikipedia.org/w/index.php?title=B%C3%A0i_to%C3%A1n_x%E1%BA%BFp_ba_l%C3%B4&action=edit&section=4)]

Bài toán xếp ba lô có thể được giải trong [thời gian giả-đa thức](https://vi.wikipedia.org/w/index.php?title=Th%E1%BB%9Di_gian_gi%E1%BA%A3-%C4%91a_th%E1%BB%A9c&action=edit&redlink=1) bằng [quy hoạch động](https://vi.wikipedia.org/wiki/Quy_ho%E1%BA%A1ch_%C4%91%E1%BB%99ng). Dưới đây là lời giải quy hoạch động cho *bài toán xếp ba lô không bị chặn*.

Gọi các chi phí là *c1*,..., *cn* và các giá trị tương ứng là *v1*,..., *vn*. Ta cần cực đại hóa tổng giá trị với điều kiện tổng chi phí không vượt quá *C*. Khi đó, với mỗi *i* ≤ *C*, đặt *A*(*i*) là giá trị lớn nhất có thể đạt được với tổng chi phí không vượt quá *i*. Rõ ràng, *A*(*C*) là đáp số của bài toán.

Định nghĩa *A*(*i*) một cách đệ quy như sau:

* *A*(0) = 0
* *A*(*i*) = max { *vj* + *A*(*i* − *cj*),*A(*i*) |* cj *≤* i *}*

Ở đây, giá trị lớn nhất của tập rỗng được lấy bằng 0. Tính dần các kết quả từ *A*(0) tới *A*(*C*), ta sẽ được lời giải. Do việc tính mỗi *A*(*i*) đòi hỏi xem xét *n* đồ vật (tất cả các giá trị này đã được tính từ trước), và có *C* giá trị của các *A*(*i*) cần tính, nên thời gian chạy của lời giải quy hoạch động là O(*nC*).

Điều này không mâu thuẫn với thực tế rằng bài toán xếp ba lô là [NP-đầy đủ](https://vi.wikipedia.org/wiki/NP-%C4%91%E1%BA%A7y_%C4%91%E1%BB%A7), do *C*, không như *n*, không thuộc mức đa thức theo độ dài của đầu vào cho bài toán. Độ dài đầu vào bài toán tỉ lệ thuận với số bit trong *C*, chứ không tỉ lệ với chính *C*.

Một giải pháp quy hoạch động tương tự cho *bài toán xếp ba lô 0-1* cũng chạy trong [thời gian giả-đa thức](https://vi.wikipedia.org/w/index.php?title=Th%E1%BB%9Di_gian_gi%E1%BA%A3-%C4%91a_th%E1%BB%A9c&action=edit&redlink=1). Cũng như trên, gọi các chi phí là *c1*,..., *cn* và các giá trị tương ứng là *v1*,..., *vn*. Ta cần cực đại hóa tổng giá trị với điều kiện tổng chi phí không vượt quá *C*. Định nghĩa một hàm đệ quy *A*(*i*, *j*) là giá trị lớn nhất có thể đạt được với chi phí không vượt quá *j* và sử dụng các đồ vật trong khoảng từ *x*1 tới *xi*.

*A*(*i*,*j*) được định nghĩa đệ quy như sau:

* f[i][j]:= giá trị lớn nhất có thể lấy được;
* v[i]:= là giá trị của đồ vật thứ i;
* w[i]:=trọng lượng của đồ vật thứ i
* NOTES: Đây là bài toán không giới hạn đồ vật
* ***f[i][j]=max(f[i-1][j],v[i]+f[i][j-w[i]])****// khi đã lấy vật thứ****i****thì có thể lấy thêm nữa nên ta có****f[i][j-w[i]]:=****đã trừ đi trọng lượng của đồ vật thứ i và vẫn có thể lấy thêm được nữa // i là đồ vật không giới hạn*
* *A*(0, *j*) = 0
* *A*(*i*, 0) = 0
* *A*(*i*, *j*) = *A*(*i* - 1, *j*) if ci > *j*
* *A*(*i*, *j*) = max(*A*(*i* - 1, *j*), vi + *A*(*i* - 1, *j* - ci)) if ci ≤ *j*

Để có lời giải, ta tính A(*n*, *C*). Để làm điều này, ta có thể dùng 1 bảng để lưu các tính toán trước đó. Cách giải này do đó sẽ chạy trong thời gian O(*nC*) và không gian O(*nC*), tuy ta có thể giảm độ phức tạp không gian xuống O(*C*) bằng một số sửa đổi nhỏ.

Câu 24: Để phát biểu bài toán ta cần nhắc lại một số khái niệm. Giả sử G = (V,E) là đơn đồ thị vô hướng với trọng số trên đỉnh c(), V. Một tập con các đỉnh của đồ thị được gọi là tập độc lập, nếu như hai đỉnh bất kỳ trong U là không kề nhau trên G.

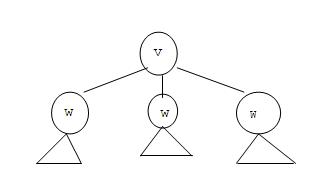
Nếu U là tập độc lập, thì ta gọi trọng số của U là tổng trọng số của các đỉnh trong nó. Ta sẽ gọi tập độc lập với trọng số lớn nhất là tập độc lập lớn nhất. Bài toán tập độc lập lớn nhất trên đồ thị là một bài toán khó. Tuy nhiên, khi đồ thị G là cây bài toán này có thể giải hiệu quả bởi thuật toán quy hoạch động.

Bài toán phát biểu như sau: Cho cây T = (V,E) với trọng số trên các đỉnh c(), V. Hãy tìm tập độc lập lớn nhất của T.

Dựng cây T có gốc tại đỉnh r, và duyệt cây theo thứ tự sau (postorder). Xét đỉnh tuỳ ý với k con w1, w2, …, wk. Ta có thể tạo tập độc lập của cây con gốc tại theo hai cách, phụ thuộc vào việc ta có chọn đỉnh vào tập độc lập hay không:

• Nếu ta không chọn vào tập độc lập, thì ta có thể kết hợp các tập độc lập của các cây con gốc tại w1, w2, …, wk để tạo tập độc lập gốc tại , bởi vì không có cạnh nối giữa các cây con này.

• Còn nếu ta chọn vào tập độc lập thì ta chỉ có thể sử dụng các tập độc lập không chứa gốc của các cây con tương ứng với w1, w2, …, wk, do và bất kỳ con wi nào của nó không cùng chọn vào tập độc lập



Do đó, với mỗi đỉnh thuật toán phải tính các thông tin sau:

1. big() = trọng lượng lớn nhất của tập độc lập của cây con có gốc tại .

2. bibnotroot() = trọng lượng lớn nhất của tập độc lập không chứa của cây con có gốc tại .

Tại đỉnh , thuật toán sẽ gọi đệ quy tính big(wi) và bignotroot(wi) với mỗi cây con gốc tại các con w1, w2, …, wk của . Sau đó tính bignotroot() và big() sử dụng công thức đệ quy tương ứng với hai tình huống mô tả ở trên:

|  |
| --- |
| bign    otroot    (    *v*    )    =    *k*  ∑  *i*    =    1    big    (    w  *i*    ) |
| big    (    *v*    )    =    max    {    bignotroot    (    *v*    )    *,*    *c*    (    *v*    )    +    *k*  ∑  *i*    =    1    bignotroot    (    w  *i*    )    } |

Nếu là lá thì bignotroot() = 0 và big() = c().

Từ các phân tích trên dễ dàng xây dựng thuật toán tính big(), V với thời gian O(n), trong đó n = .

Ta xét cách thực hiện đệ quy của thuật toán. Rõ ràng trọng số của tập độc lập lớn nhất tại đỉnh sẽ hoặc là bằng trọng lượng của tất cả các tập độc lập của các cây con gốc tại w1, w2, …, wk hoặc bằng tổng trọng lượng của và trọng lượng của các cây con gốc tại các đỉnh là cháu của . Từ đó ta có thuật toán đệ quy sau:

function MaxISTree(r);

(\* Tìm tập độc lập lớn nhất của cây con gốc tại r \*)

(\* Con(r) - danh sách các con của root \*)

*(\* Cháu(r) - danh sách các cháu của root \*)*

begin

if Con(r) = then return c(r) (\* r lµ l¸ \*)

else

begin

bignotroot: = 0;

*for w*

*Con(r) do*

bignotroot: = bignotroot + MaxISTree(w);

bigr: = c(r);

*for u*

*Chau(r) do*

bigr: = bigr + MaxISTree(u);

return max(bignotroot, bigr);

end;

end;

Lệnh gọi MaxISTree(root) (root là gốc của cây T) sẽ thực hiện thuật toán. Tất nhiên thủ tục đệ qui này là không hiệu quả. Do ở đây chỉ có O(n) bài toán con cần giải, ta có thể viết lại nó dưới dạng thủ tục đệ qui có nhớ để có được thuật toán với thời gian tính O(n).