# MỤC LỤC

## Contents

Bài 1. Sắp xếp và Tìm kiếm – tìm kiếm nhị phân ..... 5
I. Các phương pháp sắp xếp. ..... 5
1. Sắp xếp nổi bọt ..... 5
2. Sắp xếp chọn ..... 7
3.Sắp xếp trộn ..... 9
4. Sắp xếp nhanh ..... 11
5. Bài tập áp dụng ..... 14
II. Thuật toán tìm kiếm nhị phân ..... 15
1. Tìm kiếm và tìm kiếm nhị phân ..... 15
2. Tư tưởng thuật toán ..... 15
3. Minh họa thuật toán ..... 15
4. Code tham khảo ..... 16
5. Bài tập áp dụng : ..... 17
Bài 2: Các phương pháp sinh ..... 18
I. Giới thiệu ..... 18
II. Cấu trúc thuật toán sinh ..... 18
III. Giới thiệu bốn phương pháp sinh ..... 19
1. Sinh nhị phân ..... 19
2. Sinh hoán vị ..... 20
3. Sinh tổ hợp ..... 23
4. Sinh chỉnh hợp ..... 25
5. Bài tập ứng dụng ..... 26
Bài 3: Đệ quy quay lui và kỹ thuật nhánh cận ..... 27
I. Đệ quy ..... 27
1. Khái niệm về đệ quy ..... 27
2. Giải thuật đệ quy ..... 27

3. Một số ví dụ cơ bản về đệ quy ..... 28
4. Một số bài tập áp dụng ..... 30
II. Quay lui ..... 30
1. Giới thiệu về quay lui ..... 30
2. Mô hình thuật toán quay lui ..... 31
3. Các ví dụ về quay lui ..... 32
III. Một số bài toán đặc trưng sử dụng thuật toán Đệ quy – Quay lui ..... 35
1. Bài toán tìm đường đi trên ma trận ..... 35
2. Bài toán người du lịch ..... 36
3. Bài toán xếp hậu ..... 38
4. Bài toán quân mã đi tuần ..... 41
IV. Kỹ thuật nhánh cận ..... 44
1. Bài toán tối ưu và sự bùng nổ tổ hợp ..... 44
2. Mô hình kỹ thuật nhánh cận ..... 44
3. Một số bài toán sử dụng kỹ thuật nhánh cận ..... 45
Bài 4: Cấu trúc dữ liệu STACK – QUEUE ..... 50
I. STACK – Ngăn xếp ..... 50
II. QUEUE – Hàng đợi ..... 53
III. Priority_Queue_Hàng đợi ưu tiên ..... 54
1. Khái niệm: ..... 54
2. Cấu trúc Heap: ..... 55
3. Cách sử dụng priority queue trong thư viện STL: ..... 56
Bài 5 : Quy hoạch động ..... 59
I. Quy hoạch động là gì? ..... 59
1. Đệ quy và quy hoạch động : ..... 59
2. Quy hoạch động là gì ? ..... 59
3. Ưu nhược điểm : ..... 59
II. Hai cách tiếp cận quy hoạch động. ..... 59
1. Top-down (Từ trên xuống): ..... 60
2. Bottom-up (Từ dưới lên) ..... 61

3. Phương pháp giải bài toán quy hoạch động : 62
III. Một số bài toán, ví dụ về quy hoạch động 62
1. Bài toán về dãy con đơn điệu dài nhất 62
2. Bài toán xâu con chung dài nhất. 66
3. Bài toán cái túi – quy hoạch động. 69
Bài 6: Đồ thị - Các phương pháp duyệt đồ thị 72
I. Lý thuyết cơ bản về đồ thị 72
1.Định nghĩa đồ thị: 72
2. Phân loại 73
3. Các khái niệm 74
4. Các cách biểu diễn và lưu trữ đồ thị 75
II.Các thuật toán duyệt đồ thị không trọng số 81
1.Thuật toán duyệt đồ thị theo chiều sâu (Depth-First Search - DFS) 81
2. Thuật toán duyệt đồ thị theo chiều rộng (Breadth-First Search - BFS) 90
III. Bài tập áp dụng 95
TÀI LIỆU THAM KHẢO 96

# TÀI LIỆU
# THUẬT TOÁN CƠ BẢN

# Bài 1. Sắp xếp và Tìm kiếm – tìm kiếm nhị phân

## I. Các phương pháp sắp xếp.

Ta có 1 dãy số $a_1$, $a_{2,}$, …, $a_N$ được lưu trữ trong cấu trúc dữ liệu mảng. Sắp xếp dãy số $a_1$, $a_2$, …., $a_N$ là thực hiện việc bố trí lại các phần tử sao cho hình thành được dãy mới $a_{k1}$, $a_{k2}$, …., $a_{kN}$ có thứ tự (ví dụ thứ tự tăng) nghĩ là $a_{ki} > a_{ki}$ -1.

Tùy theo yêu cầu bài toán và tính chất của dãy số,ta có các phương pháp sắp xếp khác nhau. Tuy nhiên, ở đây mình sẽ giới thiệu 4 phương pháp phổ thông nhất đó là: sắp xếp nổi bọt,sắp xếp chọn, sắp xếp trộn và sắp xếp nhanh.

### 1. Sắp xếp nổi bọt

***Ý tưởng :*** Xuất phát từ đầu dãy, so sánh 2 phần tử cạnh nhau để đưa phần tử nhỏ hơn lên trước, sau đó lại xét cặp tiếp theo cho đến khi tiến về đầu dãy. Nhờ vậy, ở lần xử lý thứ i sẽ tìm được phần tử ở vị trí đầu dãy là i.

***Giải thuật :***

- Bước 1: i=1  // Lần xử lý đầu tiên
- Bước 2: j=N  // Duyệt từ cuối dãy trở về vị trí
  Trong khi (j > i) thực hiện:

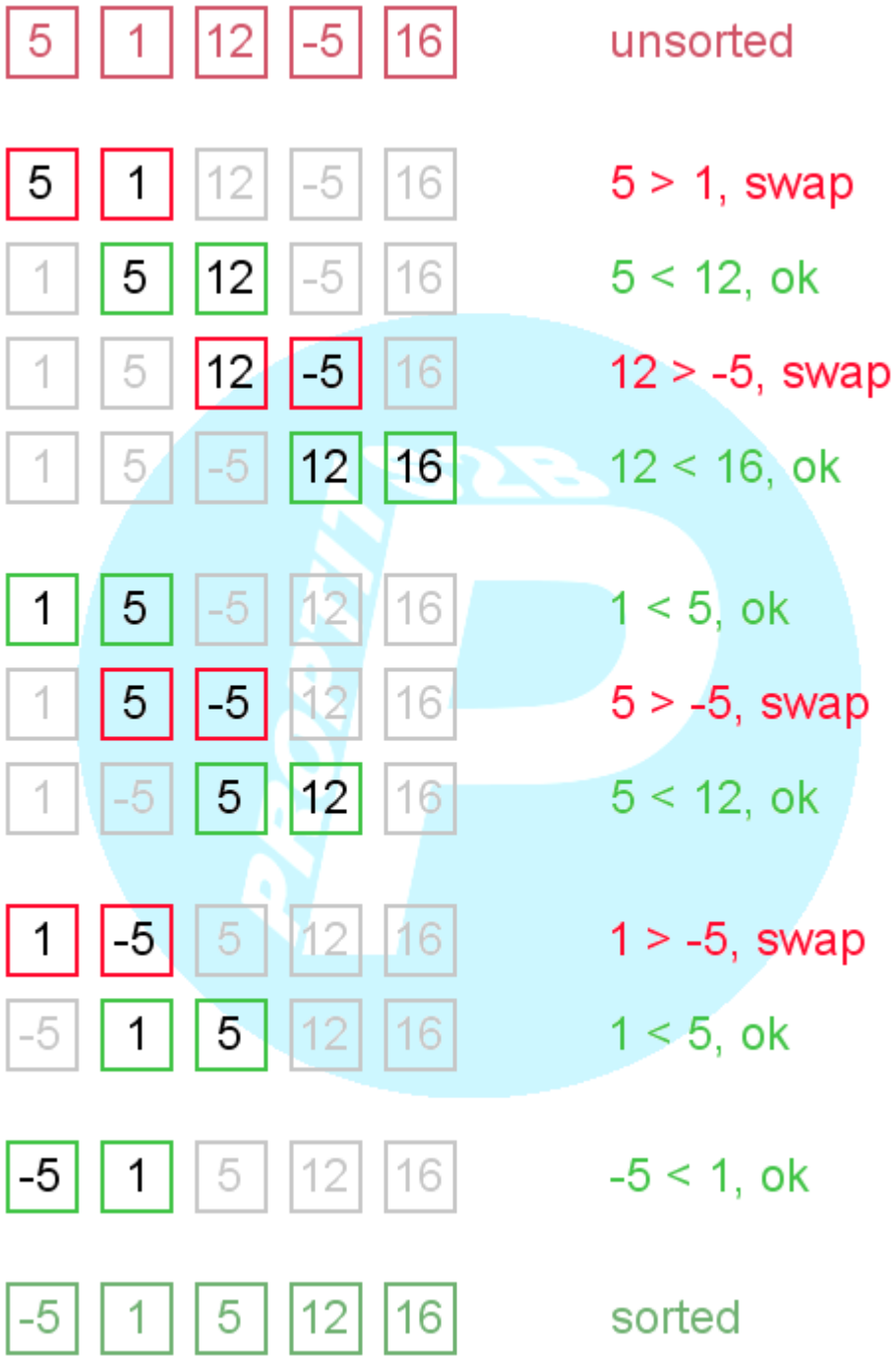
  Nếu a[j] < a[j-1]: hoán vị a[j] và a[j-1]

  j=j-1;
- Bước 3: i=i+1  // Lần xử lý tiếp theo
  Nếu I > N-1 thì dừng

  Ngược lại, lặp lại bước 2.

***Nhận xét :*** Lần duyệt đầu tiên ta cần duyệt n phần tử,lần duyệt thứ 2 cần duyệt n-1 phần tử,…cứ duyệt như vậy cho đến lần duyệt cuối cùng là 1 phần tử.Vậy số lần duyệt của phương pháp này là n + (n-1) + (n-2) + ... 2 +1 = n(n+1)/2 tương đương với độ phức tạp O(n^2).

***Ví dụ :*** Cho mảng a[5]={ 5, 1, 12, -5, 16 }. Chúng ta cần sắp xếp mảng tăng dần theo giá trị. Khi đó, thuật toán Bubble Sort sẽ được trình bày như sau:




***Code :***

```
void bubbleSort(int arr[], int n) {
      int tmp;
      for (int i = 0 ; i <= n - 1; ++i)
         for (int j = 0 ; j < n - i ; j++)
            if ( arr[j] > arr[j+1] ) {
                  tmp = arr[j];
                  arr[j] = arr[j+1];
                  arr[j+1] = tmp;
               }
}
```

## 2. Sắp xếp chọn

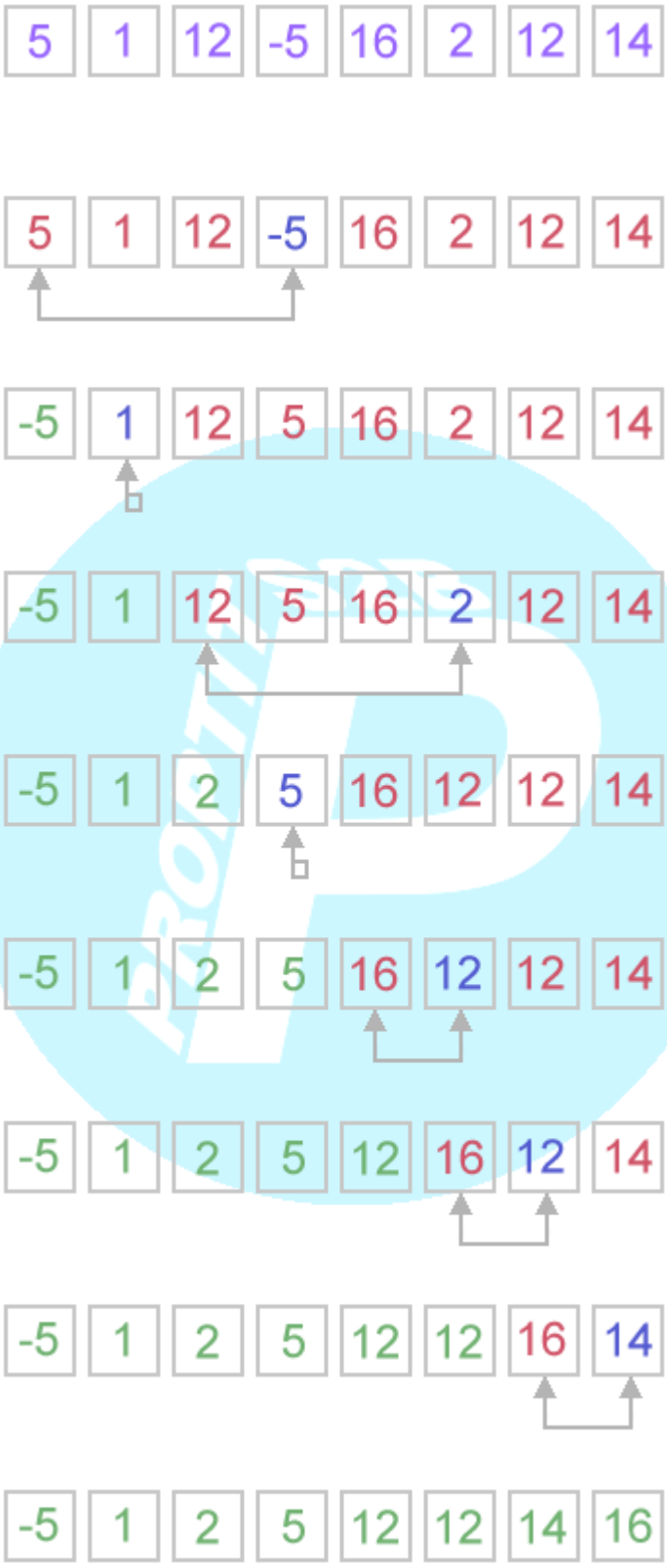
***Ý tưởng :*** Chọn phần tử nhỏ nhất trong n phần tử ban đầu, đưa phần tử này về vị trí đúng là đầu tiên của dãy hiện hành. Sau đó không quan tâm đến nó nữa, xem dãy hiện hành chỉ còn n-1 phần tử của dãy ban đầu, bắt đầu từ vị trí thứ 2. Lặp lại quá trình trên cho dãy hiện hành đến khi dãy hiện hành chỉ còn 1 phần tử.

***Giải thuật :***

- Bước 1 : Đặt i=1,min=1 với min là biến dùng để tìm vị trí phần tử nhỏ nhất trong dãy đang xét.
- Bước 2 : Tìm phần tử a[min] nhỏ nhất trong dãy đang xét từ i đến n.
- Bước 3 : Hoán vị a[min] và a[i].
- Bước 4 : Nếu i<=n-1 thì i=i+1 và quay lại bước 2.Nếu i=n thì dãy đã được sắp xếp.

***Độ phức tạp :*** O(n^2) tương đương với phương pháp nổi bọt.

***Ví dụ :*** Cho dãy a[7]= {5, 1, 12, -5, 16, 2, 12, 14} .


5 1 12 -5 16 2 12 14
5 1 12 -5 16 2 12 14
-5 1 12 5 16 2 12 14
-5 1 12 5 16 2 12 14
-5 1 2 5 16 12 12 14
-5 1 2 5 16 12 12 14
-5 1 2 5 12 16 12 14
-5 1 2 5 12 12 16 14
-5 1 2 5 12 12 14 16

***Code :***

```
void selectionSort(int arr[], int n) {
	int i, j, minIndex, tmp;
	for (i = 0; i < n - 1; i++) {
		minIndex = i;
		for (j = i + 1; j < n; j++)
			if (arr[j] < arr[minIndex])
				minIndex = j;
		if (minIndex != i) {
			tmp = arr[i];
			arr[i] = arr[minIndex];
			arr[minIndex] = tmp;
		}
	}
}
```

### 3.Sắp xếp trộn

***Ý tưởng :*** Giả sử dãy X[1],...,X[i],...,X[n] có hai đoạn đã được sắp xếp không giảm là X[1],...,X[i] và X[i+1],...,X[n], ta tiến hành tạo ra dãy mới bằng cách lần lượt lấy hai phần tử đầu tiên của mỗi đoạn và so sánh với nhau. Phần tử nào nhỏ hơn thì lấy ra dãy mới và bỏ nó ra khỏi đoạn đang xét. Cứ tiến hành như vậy cho tới khi một dãy bị vét hết (không còn phần tử nào). Lấy toàn bộ phần tử còn lại của đoạn không xét hết nối vào sau dãy đích. Như vậy dãy mới là một dãy đã được sắp xếp.

Trong mọi trường hợp, có thể coi dãy gồm duy nhất 1 phần tử là đã được sắp xếp. Lợi dụng việc này, ta phân chia một dãy ra thành nhiều dãy con chỉ gồm một phần tử rồi lần lượt hòa nhập từng đôi một với nhau.

***Ví dụ:*** Sắp xếp dãy số a[5] = {7, -5, 2, 16, 4}

| | | | | | |
|---|---|---|---|---|---|
| 7 | -5 | 2 | 16 | 4 | unsorted |
| 7 | -5 | 2 | 16 | 4 | -5 to be inserted |
| ? | 7 | 2 | 16 | 4 | 7 > -5, shift |
| -5 | 7 | 2 | 16 | 4 | reached left boundary, insert -5 |
| -5 | 7 | 2 | 16 | 4 | 2 to be inserted |
| -5 | ? | 7 | 16 | 4 | 7 > 2, shift |
| -5 | 2 | 7 | 16 | 4 | -5 < 2, insert 2 |
| -5 | 2 | 7 | 16 | 4 | 16 to be inserted |
| -5 | 2 | 7 | 16 | 4 | 7 < 16, insert 16 |
| -5 | 2 | 7 | 16 | 4 | 4 to be inserted |
| -5 | 2 | 7 | ? | 16 | 16 > 4, shift |
| -5 | 2 | ? | 7 | 16 | 7 > 4, shift |
| -5 | 2 | 4 | 7 | 16 | 2 < 4, insert 4 |
| -5 | 2 | 4 | 7 | 16 | sorted |

***Code:***

```
void insertionSort(int arr[], int length) {
	int i, j, tmp;
	for (i = 1; i < length; i++) {
		j = i;
		while (j > 0 && arr[j - 1] > arr[j]) {
			tmp = arr[j];
			arr[j] = arr[j - 1];
			arr[j - 1] = tmp;
			j--;
		}
	}
}
```

## 4. Sắp xếp nhanh

***Ý tưởng :*** Dựa trên ý tưởng chia để trị, QuickSort chia mảng thành hai danh sách bằng cách so sánh từng phần tử của danh sách với một phần tử được chọn được gọi là phần tử chốt. Những phần tử nhỏ hơn hoặc bằng phần tử chốt được đưa về phía trước và nằm trong danh sách con thứ nhất, các phần tử lớn hơn chốt được đưa về phía sau và thuộc danh sách con thứ hai. Cứ tiếp tục chia như vậy tới khi các danh sách con đều có độ dài bằng 1.

**Có các cách chọn phần tử chốt như sau:**

- Chọn phần tử đứng đầu hoặc đứng cuối làm phần tử chốt.
- Chọn phần tử đứng giữa danh sách làm phần tử chốt.
- Chọn phần tử trung vị trong 3 phần tử đứng đầu, đứng giữa và đứng cuối làm phần tử chốt.
- **Chọn phần tử ngẫu nhiên làm phần tử chốt (Cách này hay được chọn vì tránh được các trường hợp đặc biệt)**

***Giải thuật :***

***Bước 1:*** Nếu left <= right,thực hiện bước 2.Ngược lại thì kết thúc.

***Bước 2:*** Chọn $x=a_{(right-left)/2}$ là phần tử chốt,i=right,j=left.
Phân hoạch dãy bau đầu $a_{left} \dots a_{right}$ thành các đoạn : $a_{left} \dots a_j$(đoạn chứa những phần tử nhỏ hơn x),$a_{j+1} \dots a_{i-1}$ (đoạn chứa những phần tử bằng x), $a_i \dots a_{right}$ (đoạn chứa những phần tử lớn hơn x).

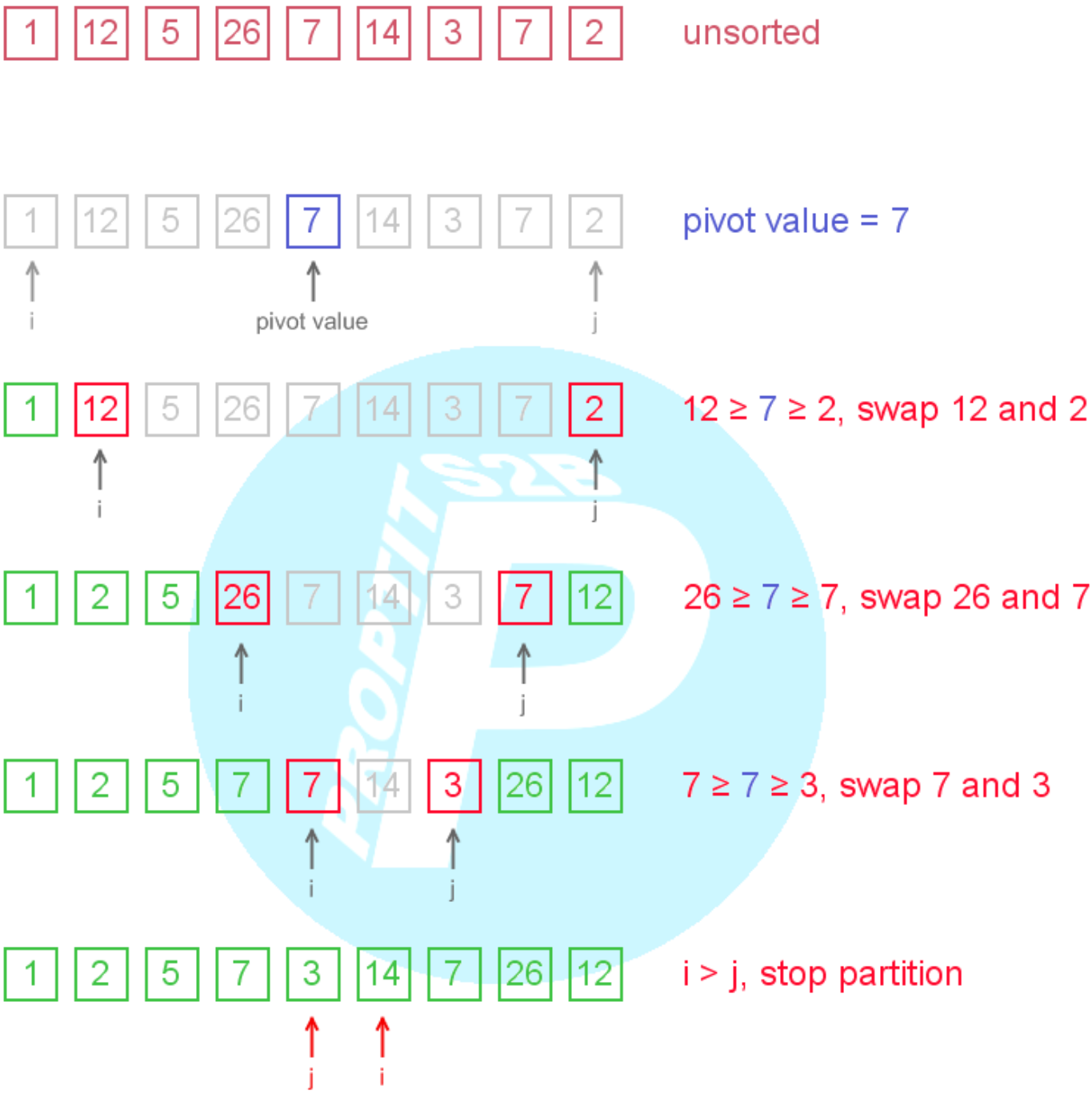
***Bước 3***: Sắp xếp đoạn 1: $a_{left}$ … $a_j$ : quay lại bước 1,với right =j.

***Bước 4:*** Sắp xếp đoạn 3: $a_i$ … $a_{right}$ : quay lại bước 1,với left=i.

***Nhận xét :*** Viêc chọn phần tử là phần tử chốt quyết định đến khả năng xử lý của phương pháp này.

- **Trường hợp tốt nhất:** mỗi lần phân hoạch ta đều chọn được phần tử trung vị (phần tử lớn hơn hay bằng nửa số phần tử và nhỏ hơn hay bằng nửa số phần tử còn lại) làm mốc. Khi đó dãy được phân hoạch thành hai phần bằng nhau, và ta cần log2(n) lần phân hoạch thì sắp xếp xong. Ta cũng dễ nhận thấy trong mỗi lần phân hoạch ta cần duyệt qua n phần tử. Vậy độ phức tạp trong trường hợp tốt nhất thuộc O(nlog2(n)).
- **Trường hợp xấu nhất:** mỗi lần phần hoạch ta chọn phải phần tử có giá trị cực đại hoặc cực tiểu làm mốc. Khi đó dãy bị phân hoạch thành hai phần không đều: một phần chỉ có một phần tử, phần còn lại có n-1 phần tử. Do đó, ta cần tới n lần phân hoạch mới sắp xếp xong. Vậy độ phức tạp trong trường hợp xấu nhất thuộc O(n2).
- Trường hợp trung bình : Độ phức tạp O(nlogn).

***Ví dụ :*** Cho mảng a[10]= {1, 12, 5, 26, 7, 14, 3, 7, 2}.


1 12 5 26 7 14 3 7 2
unsorted
1 12 5 26 7 14 3 7 2
pivot value = 7
i
pivot value
j
1 12 5 26 7 14 3 7 2
12 ≥ 7 ≥ 2, swap 12 and 2
i
j
1 2 5 26 7 14 3 7 12
26 ≥ 7 ≥ 7, swap 26 and 7
i
j
1 2 5 7 7 14 3 26 12
7 ≥ 7 ≥ 3, swap 7 and 3
i
j
1 2 5 7 3 14 7 26 12
i > j, stop partition
j
i
1 2 5 7 3
14 7 26 12
run quick sort recursively
...
1 2 3 5 7 7 12 14 26
sorted

***Code* :**

```
void quickSort(int arr[], int left, int right) {
      int i = left, j = right;
      int tmp;
      int pivot = arr[(left + right) / 2];

      while (i <= j) {
            while (arr[i] < pivot)
                  i++;
            while (arr[j] > pivot)
                  j--;
            if (i <= j) {
                  tmp = arr[i];
                  arr[i] = arr[j];
                  arr[j] = tmp;
                  i++;
                  j--;
            }
      };

      if (left < j)
            quickSort(arr, left, j);
      if (i < right)
            quickSort(arr, i, right);
}
```

**5. Bài tập áp dụng**

http://vn.spoj.com/problems/DHEXP/

http://www.spoj.com/PTIT/problems/PTIT123A/

## II. Thuật toán tìm kiếm nhị phân

### 1. Tìm kiếm và tìm kiếm nhị phân

Để tìm một giá trị trong mảng không được sắp xếp, chúng ta phải duyệt qua từng phần tử của mảng cho đến khi tìm được giá trị cần tìm, đó chính là tìm kiếm thông thường. Trong trường hợp giá trị tìm kiếm đó không có trong mảng, ta sẽ phải duyệt qua tất cả các phần tử. Độ phức tạp của thuật toán như vậy sẽ tỷ lệ thuận với chiều dài của mảng, khá là lớn. Tuy nhiên chúng ta có thể cải tiến để có thể tìm kiếm nhanh hơn rất nhiều lần khi mảng đã được sắp xếp bằng thuật toán tìm kiếm nhị phân. Cũng như cái tên của nó, tìm kiếm nhị phân với các dãy hoặc chuỗi số tuyến tính với độ phức tạp khá bé : O(log n).

### 2. Tư tưởng thuật toán

Binary Search tìm kiếm một phần tử cụ thể bằng cách so sánh phần tử tại vị trí giữa nhất của tập dữ liệu. Nếu tìm thấy kết nối thì chỉ mục của phần tử được trả về. Nếu phần tử cần tìm lớn hơn giá trị phần tử giữa thì phần tử cần tìm được tìm trong mảng con nằm ở bên phải phần tử giữa; Nếu không thì sẽ tìm ở trong mảng con nằm ở bên trái phần tử giữa. Tiến trình sẽ tiếp tục như vậy trên mảng con cho tới khi tìm hết mọi phần tử trên mảng con này.

Khi đó, chi phí của thuật toán tìm kiếm giảm xuống logarithm nhị phân của chiều dài mảng. Để tham khảo, log2 (1 000 000) ≈ 20. Điều này có nghĩa là, trong trường hợp xấu nhất, thuật toán thực hiện 20 bước để tìm một giá trị trong mảng sắp xếp của một triệu phần tử.

### 3. Minh họa thuật toán

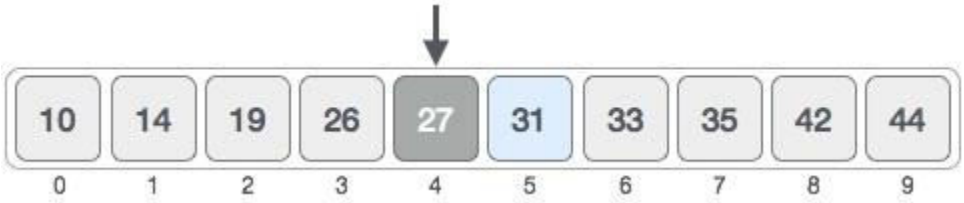
Giả sử chúng ta cần tìm vị trí của giá trị 31 trong một mảng bao gồm các giá trị như hình dưới đây bởi sử dụng Binary Search:




Đầu tiên, chúng ta chia mảng thành hai nửa, theo phép toán sau:

```
chỉ-mục-giữa = (cuối + đầu)/ 2
```

Với ví dụ trên (9 + 0)/ 2 = 4 (giá trị là 4.5). Do đó 4 là chỉ mục giữa của mảng.




Bây giờ chúng ta so sánh giá trị phần tử giữa với phần tử cần tìm. Giá trị phần tử giữa là 27 và phần tử cần tìm là 31, do đó là không kết nối. Bởi vì giá trị cần tìm là lớn hơn nên phần tử cần tìm sẽ nằm ở mảng con bên phải phần tử giữa.




Khi đó ta sẽ đổi chỉ-mục-đầu = chỉ-mục-giữa + 1 = 4 + 1 = 5 và lại tiếp tục tìm kiếm giá trị chỉ-mục-giữa. Chỉ-mục-giữa = ( 9 + 5 ) / 2 = 7.




Tiếp tục tìm chỉ-mục-giữa lần nữa. Lần này nó có giá trị là 5.




Cuối cùng ta sẽ tìm được vị trí giá trị 31 trong dãy trên trong log2(10) xấp xỉ bằng 3 bước.

## 4. Code tham khảo

Đoạn code sau áp dụng cho bài toán tìm vị trí của giá trị value trong dãy arr[] bằng tìm kiếm nhị phân :

```
int binarySearch(int arr[], int value, int left, int right) {
	while (left <= right) {
		int middle = (left + right) / 2;
		if (arr[middle] == value)
			return middle;
		else if (arr[middle] > value)
			right = middle - 1;
		else
			left = middle + 1;
	}
	return -1;
}
```

**5. Bài tập áp dụng :**

http://www.spoj.com/PTIT/problems/P155SUMD/

http://www.spoj.com/problems/NDCCARD/

http://vn.spoj.com/problems/CRUELL2/

# Bài 2: Các phương pháp sinh

## I. Giới thiệu

Trong Toán học Tổ hợp, cấu hình là một nghiệm bất kỳ thỏa mãn yêu cầu của bài toán tổ hợp. Mọi cấu hình đều là một tập hợp các phần tử cho bài toán.

Ví dụ: Cho tập hợp X = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19}. Liệt kê một số chỉnh hợp không lặp chập 3 của tập hợp.

Với bài toán như trên, một số cấu hình có thể là {1, 4, 11}, {4, 1, 7}, {13, 19, 2}.

Ta nhận thấy, nếu mở rộng ra với X.size() và bậc chỉnh hợp rất lớn, bài toán sẽ không thể đơn giản được giải quyết bởi việc đếm theo cảm tính – phải có một quy luật để liệt kê toàn bộ các cấu hình thỏa mãn. Do đó, việc phát triển các thuật toán sinh trở nên cần thiết.

Một chương trình muốn sử dụng được thuật toán sinh phải thỏa mãn được 2 điều kiện sau:

1. Xác định trình tự, điểm đầu và điểm cuối của tập hợp cấu hình.Thật vậy, trong khi điểm đầu có ý nghĩa là cơ sở để hàm sinh chạy được, điểm cuối của tập hợp cấu hình sẽ đảm bảo thuật toán có điểm dừng.
2. Xây dựng được thuật toán mà từ một cấu hình chưa phải cuối thì ta luôn sinh ra một cấu hình kế tiếp nó.

Hiển nhiên, muốn sinh ra được toàn bộ các cấu hình thì ta phải có một thuật toán để biến đổi có quy tắc các cấu hình nhằm tạo được cấu hình mới.

## II. Cấu trúc thuật toán sinh

Phương pháp sinh có thể mô tả như sau:

```
<Xây dựng cấu hình đầu tiên>;

While(chưa phải cấu hình cuối)
{
     <Đưa ra cấu hình đang có>;
     <Từ cấu hình đang có sinh ra cấu
hình kế tiếp nếu còn>;
}
```

## III. Giới thiệu bốn phương pháp sinh

### 1. Sinh nhị phân

- Một dãy nhị phân có độ dài n là một dãy x=x1x2x3...xn trong đó xi € {0,1} (i : 1≤i≤n).

- Như vậy cấu hình bắt đầu là: 000...0 (n số 0) , và cấu hình kết thúc là : 111...1 (n số 1).

- ***Ví dụ*** : Dãy nhị phân độ dài 3 bao gồm {000,001,010,011,100,101,110,111}
- ***Phân tích vấn đề:***

Ta nhận thấy: Số chuỗi nhị phân sinh ra được là $2^n$, chuỗi đầu tiên là 00…0 và chuỗi cuối cùng là 11…1. Nhận thấy nếu chuỗi hiện tại chưa phải chuỗi cuối cùng cần liệt kê thì chuỗi tiếp theo sẽ bằng chuỗi hiện tại cộng thêm một ( theo hệ cơ số 2 có nhớ ) .

Như vậy phương pháp sinh thỏa mãn 2 điều kiện : biết được cấu hình đầu cuối và thuật toán sinh được cấu hình tiếp theo từ cấu hình trước đó.

Phương pháp sinh được áp dụng cho bài này như sau : Xét từ cuối dãy lên gặp số 0 đầu tiên ta thay số 0 đó bằng số 1 và đặt tất cả các phần tử sau nó là 0 làm như vậy cho tới khi không tìm thấy số 0 nào ( cấu hình cuối cùng ).

***Thuật toán sinh nhị phân :***

• Bước 1 : Khởi tạo cấu hình ban đầu : 000...0 (n số 0)

• Bước 2 : Xét từ cuối dãy về đầu, gặp số 0 đầu tiên thì thay nó bằng số 1 và đặt tất cả các phần tử phía sau vị trí đó bằng 0.

• Bước 3 : In cấu hình và quay lại bước 2.Sinh cho đến khi gặp cấu hình cuối : 111...1 (n số 1) thì kết thúc quá trình sinh.

***Code :***

```
#include<iostream>
using namespace std;
int a[100],n,stop =0;
void khoitao()
{
      for(int i =0;i<n;i++)
a[i]=0;
}
void xuat()
{
      for(int i=0;i<n;i++)
cout<<a[i]<<" ";
      cout<<endl;
}
void sinh()
{
      int i=n-1;
     while(i>=0&& a[i]==1)
     {
          a[i]=0;
          i--;
     }
     if(i==-1) stop =1;
     else a[i]=1;
}
```

```
void ctrinh()
{
     stop =0;
     while(stop==0)
     {
          xuat();
          sinh();
     }
}
main()
{
     cin>>n;
     khoitao();
     ctrinh();
}
```

## 2. Sinh hoán vị

Một hoán vị là một dãy có thứ tự chứa mỗi phần tử của một tập hợp một và đúng một lần.

Điểm khác nhau cơ bản giữa một hoán vị và một tập hợp là: những phần tử của một hoán vị được sắp xếp theo một thứ tự xác định.

***Ví dụ***: liệt kê các hoán vị của {1, 2, …, n} theo thứ tự từ điển,với n=3 ta có các hoán vị : {123,132,213,231,312,321}.

***Phân tích vấn đề :***

Ta nhận thấy cấu hình đầu tiên là {1, 2, 3, …, n} và cấu hình cuối sẽ là {n, n-1, n-2, …,1}.

Hoán vị sinh ra phải lớn hơn hoán vị hiện tại hơn thế nữa phải là hoán vị vừa đủ lớn hơn hoán hoán vị hiện tại theo nghĩa không thể có một hoán vị nào khác chen giữa chúng khi sắp thứ tự.

Giả sử hoán vị hiện tại là x = {3, 2, 6, 5, 4, 1}, xét 4 phần tử cuối cùng, ta thấy chúng được sắp xếp giảm dần, điều đó có nghĩa là cho dù ta có hoán vị 4 phần tử này thế nào ta cũng chỉ thu được hoán vị nhỏ hơn hoán vị hiện tại. Như vậy ta phải xét đến x[2] = 2, thay nó bằng giá trị khác. Câu hỏi đặt ra là ta sẽ thay nó bằng giá trị nào ? Không thể thay bằng 1 vì nếu vậy sẽ được hoán vị nhỏ hơn, không thể là 3 vì đã có x[1] = 3 rồi (phần tử đứng sau không được chọn vào giá trị mà phần tử trước đã chọn ). Còn lại các giá trị 4, 5, 6. Vì cần một hoán vị vừa đủ lớn hơn hiện tại nên ta chọn x[2] = 4. Còn các giá trị còn lại sẽ lấy trong tập {2, 6, 5, 1}. Cũng vì tính vừa đủ lớn nên ta sẽ tìm biểu diễn nhỏ nhất của bốn số này gán cho x[3], x[4], x[5], x[6] tức là {1, 2, 5, 6}. Vậy hoán vị mới sẽ là {3, 4, 1, 2, 5, 6}.

***Nhận xét :*** Đoạn cuối của hoán vị hiện tại được sắp xếp giảm dần, số x[5]= 4 là số nhỏ nhất thỏa mãn điều kiện lớn hơn x[2]= 2. Nếu đổi chỗ x[5] cho x[2] và đoạn cuối vẫn được sắp xếp giảm dần thì ta sẽ được hoán vị tiếp theo của hoán vị hiện tại.

Vậy kĩ thuật sinh hoán vị kế tiếp từ hoán vị hiện tại được xây dựng như sau :

Xác định đoạn cuối giảm dần dài nhất, tìm chỉ số i của phần tử x[i] đứng liền trước đoạn cuối đó. Điều này đồng nghĩa với việc tìm từ vị trí sát cuối lên đầu, gặp chỉ số đầu tiên thỏa mãn x[i]<x[i+1] .

- Nếu tìm thấy chỉ số i như trên
  - Trong đoạn cuối giảm dần, tìm phần tử x[k] nhỏ nhất thỏa mãn điều kiện x[k] >x[i].
  - Đảo giá trị x[k] và x[i]
  - Sắp xếp đoạn cuối trở thành tang dần
- Nếu không tìm thấy tức là toàn dãy đã sắp xếp giảm dần, đây là cấu hình cuối cùng

***Thuận toán sinh hoán vị :***

• Bước 1 : Khởi tạo cấu hình ban đầu : 1,2,3,..n.

• Bước 2 : Xét từ cuối dãy lên đầu dãy,tìm phần tử đầu tiên làm mất tính không giảm của dãy,đánh dấu phần tử đó.Nếu không tìm được,kết thúc quá trình sinh.

• Bước 3 : Xét từ cuối lên đầu dãy,tìm phần tử đầu tiên lớn hơn phần tử được dánh dấu,đổi chỗ 2 phần tử đó.

• Bước 4 : Từ vị trí phần tử tìm được ở bước 2,sắp xếp tăng dần từ vị trí đó cho đến cuối dãy.In cấu hình và quay lại bước 2.

***Code :***

```
#include<iostream>
using namespace std;
int n,stop=0,a[9];
void khoitao()
{
      for(int i=1;i<=n;i++)
      a[i]=i;
}
void sinh()
{
    int i=n-1;
    while(i>0&&a[i]>a[i+1])
i--;
    if(i==0) stop=1;
   else
   {
        int k=n;
        while(a[i]>a[k])
k--;
        swap(a[k],a[i]);
        int c=n,r=i+1;
        while(r<c)
        {
            swap(a[c],a[r]);
            r++;c--;
        }
   }

}
```

```
void in()
{
     for(int i=1;i<=n;i++)
          cout<<a[i];
     cout<<endl;

}
void hoanvi()
{

    Do
   {
       in();
       sinh();
   }
   while(!stop);
}
main()
{
    cin>>n;
    khoitao();
    hoanvi();
}
```

## 3. Sinh tổ hợp

***Tổ hợp :*** Là các tập con k phần tử của n phần tử ban đầu,không có sự phân biệt về thứ tự.

***Sinh tổ hợp :*** Là tạo ra tất cả các tổ hợp (Tập con có k phần tử) của một tập hợp n phần tử.Vì mỗi tổ hợp là một tập hợp con, do đó việc liệt kê tất cả các tổ hợp của một tập hợp tương đương với việc liệt kê tất cả các tập hợp con của 1 tập hợp.

***Ví dụ :*** tổ hợp chập 3 của dãy {1,2,3,4} ta có [{1,2,3};{1,2,4};{1,3,4};{2,3,4}]

***Phân tích vấn đề:***

Ta nhận thấy tập con đầu tiên ( cấu hình khởi tạo ) là {1, 2, 3, …, k} và cấu hình kết thúc là { n-k+1, n-k+2, n-k+3, …,n} .

Ta sẽ in ra tập con bằng cách in ra tập con bằng cách in ra lần lượt các phần tử của nó theo thứ tự tăng dần . Biểu diễn mỗi tập con là một dãy x[ 1…k] trong đó x[1] < x[2] < …< x[k]. Ta nhận thấy giới hạn trên ( giá trị lớn nhất có thể nhận) của x[k] là n, của x[k-1] là n-1, của x[k-2] là n-2. Tổng quát giới hạn trên của x[i] = n-k+i , giới hạn dưới của x[i] = x[i-1] +1.

Như vậy nếu ta đang có một dãy x đại diện cho một tập con , nếu x là cấu hình kết thúc có nghĩa là tất cả các phần tử trong x đều đã đạt tới giới hạn trên thì quá trình sinh kết thúc, nếu không thì ta phải sinh ra một dãy x mới  tăng dần thỏa mãn vừa đủ lớn hơn dãy cũ theo nghĩa không có một tập con k phần tử nào chen giữa chúng khi sắp xếp thứ tự từ điển.

***Ví dụ :*** n = 9, k = 6. Cấu hình đang có x ={1, 2, 6, 7, 8, 9} . Các phần tử từ x[3] đến x[6] đã đạt tới giớn hạn trên nên để sinh cấu hình mới ta không thể sinh bằng cách tăng một phần tử trong đoạn đó lên được , ta phải tăng x[2] lên thành 3. Được cấu hình mới là x ={1, 3, 6, 7, 8, 9}. Cấu hình này thỏa mãn lớn hơn cấu hình trước nhưng chưa thỏa mãn tính chất vừa đủ lớn vậy ta lại thay x[3], x[4], x[5], x[6] bằng các giới hạn dưới của nó. Ta được cấu hình x ={1, 3, 4, 5, 6, 7} là cấu hình kế tiếp. Nếu muốn tìm tiếp, ta lại nhận thấy rằng x[6]= 7 chưa đạt tới giá trị hạn trên , như vậy chỉ cần tang x[6] lên 1 là được x = {1, 3, 4, 5, 6, 8}.

***Ta  có thể tóm tắt lại kĩ thuật sinh tập con kế tiếp từ tập đã có x như sau:***

> Tìm từ cuối dãy lên đầu cho tới khi gặp một phần tử x[i] chưa đạt giới hạn trên n-k+i.

Nếu tìm thấy :

- Tăng x[i] đó lên 1.
- Đặt tất cả cả các phần tử phía sau x[i] bằng giới hạn dưới.

Nếu không tìm thấy tức là mọi phần tử đã đạt giới hạn trên, đây là cấu hình cuối cùng.

***Thuật toán sinh tổ hợp :***

• Bước 1 : Khởi tạo cấu hình ban đầu.

• Bước 2 : Xét dãy từ cuối lên đầu,tìm phần tử thứ i đầu tiên không thỏa mãn a[i]=n - k + i.Nếu không tìm được,kết thúc quá trình sinh.

• Bước 3 : Tăng phần tử i lên 1 đơn vị, và từ phần tử i, toàn bộ phần tử sau nó có dạng a[i+1] =a[i] + 1 cho đến phần tử cuối.In cấu hình và quay lại bước 2.

***Code :***

```
#include<iostream>

using namespace std;
int a[100],n,k,stop=0;
void khoitao()
{
     for(int i=1;i<=k;i++)
     a[i]=i;
}
void in()
{
    for(int i=1;i<=k;i++)
         cout<<a[i]<<" ";
    cout<<endl;
}
void sinh()
{
    int i=k;
    while(a[i]==n-k+i)   i--
;
    if(i==0) stop=1;
    else a[i]++;
```

```
void tohop()
{
     while(stop==0)
     {
          in();
          sinh();
     }
}
main()
{
     cin>>n>>k;
     khoitao();
     tohop();
}
```

```
    int p=a[i];
    while(i<=k) a[i++]=p++;
}
```

## 4. Sinh chỉnh hợp

***Chỉnh hợp :*** Là các tập con k phần tử của n phần tử ban đầu,có sự phân biệt về thứ tự các phần tử giữa mỗi tập con.

***Sinh chỉnh hợp :*** Là tạo ra tất cả các tổ hợp (Tập con có k phần tử) có tính sắp xếp của một tập hợp n phần tử.

***Ví dụ :*** Chỉnh hợp chập 2 của {1,2,3} : [{1,2};{1,3};{2,1};{2,3};{3,1};{3,2}]

***Thuật toán sinh chỉnh hợp :*** tương tự sinh tổ hợp,nhưng sau mỗi cấu hình tìm được ta sinh hoán vị của cấu hình đó.

***Code :***

```
#include<iostream>
using namespace std;
int
a[100],b[10],n,k,stop=0,stop1=0;
void khoitao()
{
     for(int i=1;i<=k;i++)
    {
          a[i]=i;
          b[i]=i;
    }
}
void in()
{
     for(int i=1;i<=k;i++)
          cout<<a[b[i]]<<" ";
     cout<<endl;
}
void sinhhv()
{
     int i=k-1,m=k;
     while(i>0&&b[i]>b[i+1])
i--;
```

```
void sinh()
{
     int i=k;
     while(a[i]==n-k+i)
i--;
     if(i==0) stop=1;
     else a[i]++;
     int p=a[i];
     while(i<=k)
a[i++]=p++;
}
void next()
{
     while(stop==0)
     {
          while(stop1==0)
          {
               in();
               sinhhv();
          }
          stop1=0;
          for(int
i=0;i<=k;i++)    b[i]=i;
```

```
      while(b[m]<b[i]) m--;
      if(i==0) stop1=1;
      swap(b[m],b[i]);
      m=k;
      i++;
      while(i<m)
     {
          swap(b[i],b[m]);
          i++;
          m--;

     }
}
```

```
          sinh();
     }

}
main()
{
      cin>>n>>k;
      khoitao();
      next();
}
```

## 5. Bài tập ứng dụng

1.Sinh nhị phân cơ bản :

**http://www.spoj.com/PTIT/problems/BCSINH/**

2.Đi xem phim :

http://www.spoj.com/PTIT/problems/BCCOW/

_Gợi ý : Sinh nhị phân dãy có độ dài n để kiểm soát tất cả các trường hợp.Vs mỗi cấu hình tìm được lấy tổng khối lượng rồi so sánh vs khối lượng chiếc xe để tìm khối lượng lớn nhất.

3.Gia vị

http://www.spoj.com/PTIT/problems/P134SUMG/

_Gợi ý : Sinh nhị phân độ dài n để lọc tất cả trường hợp.

4. http://www.spoj.com/PTIT/problems/PTITSU1C

# Bài 3: Đệ quy quay lui và kỹ thuật nhánh cận

## I. Đệ quy

### 1. Khái niệm về đệ quy

Một đối tượng được gọi là ***đệ qui*** nếu nó được mô tả thông qua định nghĩa của chính nó. Nghĩa là, các đối tượng này được định nghĩa một cách *quy nạp* từ những khái niệm đơn giản nhất *cùng dạng* với nó. Trong toán học và tin học có rất nhiều đối tượng như thế.

Những bài toán ***đệ qui*** có thể được phân rã thành các bài toán nhỏ hơn, đơn giản hơn nhưng có cùng dạng với bài toán ban đầu. Những bài toán nhỏ lại được phân rã thành các bài toán nhỏ hơn. Cứ như vậy, việc phân rã chỉ dừng lại khi bài toán con đơn giản đến mức có thể suy ra ngay kết quả mà không cần phải phân rã nữa. Ta phải giải tất cả các bài toán con rồi kết hợp các kết quả đó lại để có được lời giải cho bài toán lớn ban đầu. Cách phân rã bài toán như vậy gọi là "**chia để trị**" (devide and conquer), là một dạng của đệ quy.

***Ví dụ:***

Hàm tính n! (giai thừa)

0!=1;

n! =n*(n-1)!;

Tính 10!:

10! = 10.9! và nhiệm vụ của ta là tính 9!
9! = 9.8! và nhiệm vụ của ta là tính 8!
.....
2! = 2.1! và nhiệm vụ của ta là tính 1!
1! = 1.0! và nhiệm vụ của ta là tính 0!
0! lại bằng 1, tính đc 0! ta tính đc 1!=>2!=> ...=>8!=>9!=> 10!

### 2. Giải thuật đệ quy

- Nếu lời giải của một bài toán P được thực hiện bằng lời giải của bài toán P' có dạng giống như P thì đó là một lời giải đệ quy.

Giải thuật tương ứng với lời giải như vậy gọi là giải thuật đệ quy. Nhưng cần lưu ý là : P' tuy có dạng như P nhưng nó phải *nhỏ hơn* P, dễ giải hơn P, và việc giải nó *không cần* đến P .

- Hàm hay thủ tục đệ quy gồm 2 phần:

- **Phần neo:** bài toán con nhỏ nhất, đơn giản nhất, có thể giải trực tiếp không cần qua một bài toán con nào cả

- **Phần đệ quy:** nếu bài toán hiện tại chưa thể giải được nhờ phần neo, thì ta xác định bài toán con của nó và gọi đệ quy để giải bài toán con đó, sau khi có lời giải của các bài toán con rồi thì phối hợp lại để giải bài toán hiện tại.

**- Phần đệ quy** thể hiện tính quy nạp của lời giải**. Phần neo** thể hiện tính **dừng** của lời giải (do khi gặp phần neo, lời giải sẽ dừng lại không đi sâu thêm)

- Để xây dựng được hàm đệ quy, ta phải tìm được phần neo, nghĩa là đi giải tất cả các bài toán con nhỏ nhất (như tính n! thì bài toán nhỏ nhất chính là 0! = 1), và *công thức* tính hàm phụ thuộc vào chính hàm đó (ví dụ  n! có thể tính được từ công thức n.(n-1)!).

## 3. Một số ví dụ cơ bản về đệ quy

### a. Hàm tính n! :

Giả sử ta có một hàm tính n! như sau:

```
6  int giaithua(int n){
7      if (n==0) return 1;
8      else return n*giaithua(n-1);
9  }
```

- **Phần neo**: với n = 0 hàm sẽ trả về ngay giá trị 1
- **Phần đệ quy**: với n > 0, hàm sẽ trả về n.giaithua(n-1): giả sử n-1 > 0, chưa phải là neo nên hàm sẽ đi sâu vào tính giaithua(n-1)

Để tính được giaithua(n-1), hàm lại đi sâu tính giaithua(n-2)…

Cho đến khi gặp neo: giaithua(0), hàm sẽ dừng không tiếp tục đi sâu nữa mà dừng lại và trả về kết quả cho các bài toán lớn hơn: giaithua(1) -> giaithua(2)->…->giaithua(n)

**Áp dụng với n = 3:**

- Gọi giaithua(3): máy sẽ nhớ giaithua(3) = 3.giaithua(2)

-> gọi giaithua(2): máy sẽ nhớ giaithua(2) = 2.giaithua(1)

-> gọi giaithua(1): máy sẽ nhớ giaithua(1) = 1.giaithua(0)

-> gọi giaithua(0): trả về 1

Bây giờ quay lại tính được:

-> giaithua(1) = 1.giaithua(0)  = 1.1 = 1

-> giaithua(2) = 2.giaithua(1)  = 2.1 = 2

-> giaithua(3) = 3.giaithua(2)  = 3.2 = 6

-> Kết thúc hàm.

### b. Hàm tính a^n

Bây giờ ta sẽ xem một ví dụ phức tạp hơn một chút, đó là tính a^n (a mũ n). Để ý rằng bài toán tính a^n có thể làm dễ dàng bằng cách duyệt nhân số a làm n lần. Nhưng với **n lớn**,

việc duyệt như vậy sẽ *tốn kém* thời gian. Khi đó có một giải thuật đệ quy cho kết quả *nhanh hơn*
Công việc của ta bây giờ là tìm *phần neo* và *công thức đệ quy* của bài toán.
Ta có:
Nếu n=0: a^0 = 1
Với n>0:

a^n = a*a*…*a (n số a)
= [a*a*…*a(n/2 số a)] * [a*a*…*a(n/2 số a)] nếu n **chẵn**
= a * [a*a*…*a(n/2 số a)] * [a*a*…*a(n/2 số a)] nếu n **lẻ**

Để tính a^n, duyệt trâu sẽ mất n lần duyệt, **độ phức tạp** là O(n). Nếu tính theo công thức trên, sẽ chỉ mất O(logn) do mỗi lần ta đều chia n đi 2 để tính bài toán con. Nếu n cỡ 10^18 thì log2(n) xấp xỉ 60 (rất nhanh).
Từ phân tích trên ta có:

- **Phần neo:** n = 0 thì a^0 = 1
- **Công thức** đệ quy với n>0: a^n = a^(n/2) * a^(n/2) nếu n **chẵn**
  = a* a^(n/2) * a^(n/2) nếu n **lẻ**

Từ đó ta xây dựng được hàm đệ quy tính a^n như sau:

```
10  int mu(int a,int n){
11      if (n==0) return 1;
12      int t = mu(a,n/2);
13      if (n%2==0) return t*t;
14      else return a*t*t;
15  }
```

***Ví dụ:*** Tính 2^5:

Gọi hàm mu(2,5):
=> t = mu(2,2), n = 5 lẻ: máy nhớ mu(2,5) = 2*t*t = 2*mu(2,2)*mu(2,2)

Gọi hàm mu(2,2):
=> t = mu(2,1), n = 2 chẵn: máy nhớ mu(2,2) = t*t = mu(2,1)*mu(2,1)

Gọi hàm mu(2,1):
=> t = mu(2,0), n = 1 lẻ: máy nhớ mu(2,1) = 2*t*t = 2* mu(2,0)*mu(2,0)
Gọi hàm mu(2,0):
=> gặp neo n=0 trả về 1

Sau đó quay lại tính các bài toán lớn hơn:

=> mu(2,1) = 2* mu(2,0)*mu(2,0) = 2*1*1 = 2

=> mu(2,2) = mu(2,1)*mu(2,1) = 2*2 = 4

=> mu(2,5) = 2*mu(2,2)*mu(2,2) = 2*4*4 = 32

Kết thúc hàm.

***Ưu điểm và nhược điểm của đệ quy:***

- **Ưu điểm**: dễ xây dựng, code đơn giản dễ hiểu
- **Nhược điểm**: tốn bộ nhớ, độ sâu đệ qui (Số lần gọi đệ qui) hữu hạn

### 4. Một số bài tập áp dụng

http://vn.spoj.com/PTIT/problems/BCFACT/
http://www.spoj.com/PTIT/problems/BCFIBO/
http://www.spoj.com/PTIT/problems/P144PROC/

## II. Quay lui

### 1. Giới thiệu về quay lui

Thuật toán *quay lui* dùng để *liệt kê* các *cấu hình*. Mỗi cấu hình được xây dựng bằng cách xây dựng *từng phần tử* , mỗi phần tử được chọn bằng cách ***thử*** tất cả các **khả năng** .
Giả sử cấu hình cần liệt kê có dạng x[1..n], khi đó thuật toán quay lui thực hiện qua các bước:
1) Xét tất cả các giá trị x[1] có thể nhận, thử cho x[1] nhận lần lượt các giá trị đó. Với mỗi giá trị thử gán cho x[1] ta sẽ:
2) Xét tất cả các giá trị x[2] có thể nhận, lại thử cho x[2] nhận lần lượt các giá trị đó. Với mỗi giá trị thử gán cho x[2] ta lại thử các khả năng cho x[3].. cứ tiếp tục như vậy đến bước:
...
n) Xét tất cả các giá trị x[n] có thể nhận, thử cho x[n] nhận lần lượt các giá trị

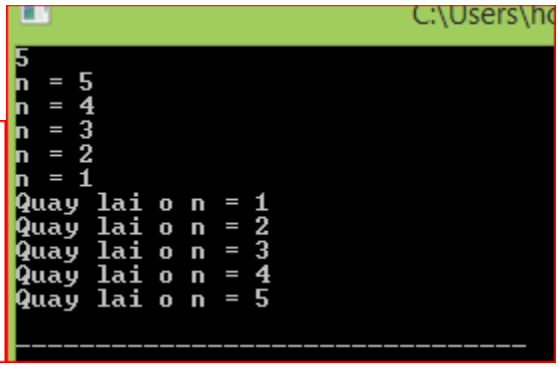
đó, thông báo cấu hình tìm được (x[1], x[2],..., x[n]).

Trên phương diện quy nạp, có thể nói rằng, thuật toán quay lui liệt kê các cấu hình n phần từ dạng x[1..n] bằng cách **thử** cho x[1] nhận lần lượt các giá trị có thể. Với mỗi giá trị thử gán cho x[1] bài toán trở thành liệt kê tiếp cấu hình n-1 phần tử x[2..n].

Quay lui cũng là **đệ quy** nhưng có thêm **phần phía sau** phần đệ quy, sẽ *quay lui* lại thực hiện sau khi đi sâu vào phần đệ quy.

Để hiểu rõ hơn về quay lui, ta xem đoạn code và kết quả sau:

```
6  void Try(int n){
7      if (n==0) return;
8      cout<<"n = "<<n<<endl;
9      Try(n-1);
10     cout<<"Quay lai o n = "<<n<<endl;
11 }
```




Ta thấy sau phần đệ quy **Try(n-1)** còn có thêm 1 dòng lệnh in ra. Dòng lệnh đó chính là phần sẽ được *quay lui* lại để thực hiện sau khi đệ quy *đi sâu* và *trở về*.

Giải thích kết quả chạy chương trình:

Nhập n = 5. Gọi Try(5), do 5>0 nên in ra n = 5, đi vào hàm Try(4) ngay mà chưa thực hiện dòng lệnh in ra cuối.
-> Gọi Try(4): n = 4>0 nên in ra 4, đi vào Try(3), chưa thực hiện dòng lệnh in ra ở cuối
…
-> Cứ thế sẽ in ra n = 5, n=4, .., n=1 như hình.
Cho đến khi gọi Try(0): thoát Try(0). Lúc này mới trở về Try(1) và thực hiện câu lệnh cout<<"Quay lai o n = "<<n<<endl;
Do lúc này n = 1 nên in ra "Quay lai o n = 1".
-> Tiếp tục quay lui về Try(2) và thực hiện nốt câu lệnh cuối. Do lúc này n = 2 nên in ra "Quay lai o n = 2".
…
-> Cứ như vậy sẽ quay lui về n = 5 và in ra "Quay lai o n = 5".

Ta nhận thấy n **càng lớn** thì dòng "Quay lai o n = …" sẽ in ra **càng lâu** phía sau, lý do là Try(n) đc gọi đầu tiên nên nó sẽ đi **sâu nhất**, và phần **quay lui** của nó sẽ phải thực hiện sau khi đã đi sâu xuống hết cỡ từ Try(n-1) cho đến Try(0) và từ Try(0) mới trở về lại Try(n).

## 2. Mô hình thuật toán quay lui

```
void Try(int i){
        for (mọi giá trị j có thể gán cho x[i]){
            <thử cho x[i] = j>;
            <đánh dấu đã chọn x[i] nếu cần>;
            if (x[i] là phần tử cuối của cấu hình) {
                <thông báo cấu hình tìm được>;
```

```
            }
            else Try(i+1);//gọi đệ quy để chọn tiếp x[i+1]
            <xóa đánh dấu đã chọn x[i] nếu cần>;
        }
}
```

Thuật toán quay lui bắt đầu với lời gọi Try(1).

## 3. Các ví dụ về quay lui

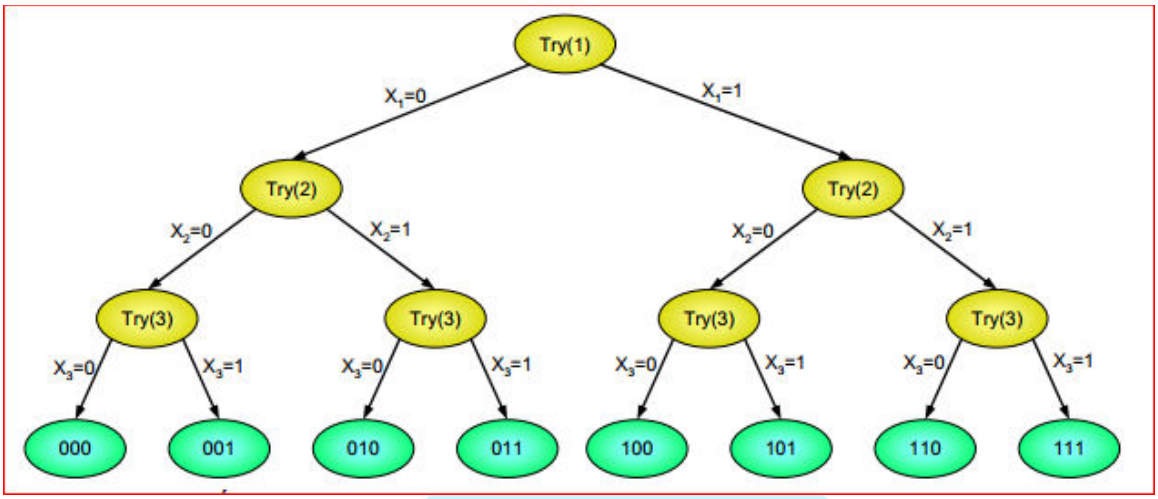
### a. Liệt kê các dãy nhị phân độ dài n

Ví dụ đơn giản nhất về quay lui là liệt các dãy nhị phân độ dài n. Ta nhận thấy mọi phần tử của cấu hình đều nhận một trong 2 giá trị 0 hoặc 1, và cách chọn phần tử hiện tại là *tùy ý*, không ảnh hưởng đến các phần tử *kế tiếp*, nên ta **không cần** phải đánh dấu là đã chọn phần tử nào đó. Dựa vào mô hình chung, ta có thể viết thủ tục **Try(i)** giải quyết bài toán như sau:

```
5   int n,x[100];
6   void xuat(){
7       for (int i=1;i<=n;i++) cout<<x[i];
8       cout<<endl;
9   }
10  void Try(int i){
11      for (int j=0;j<=1;j++){
12          x[i]=j;
13          if (i==n) xuat();
14          else Try(i+1);
15      }
16  }
```

Việc của chúng ta bây giờ là nhập độ dài n và gọi Try(1).

Có thể mô phỏng quá trình Try bằng cây sau với n = 3:




Nhìn vào cây các bạn có thể dễ dàng hiểu được cách hoạt động của thuật toán.

Ví dụ với 2 cấu hình đầu tiên: 000 và 001

Gọi Try(1): gán x[1] = 0-> đi sâu gọi Try(2), gán x[2] = 0 -> đi sâu gọi Try(3), gán x[3]=0-> (i==n) nên in ra cấu hình 000.

Bây giờ quay lui lại Try(2) tiếp tục thực hiện vòng for còn dở, gán x[2] = 1-> đi sâu gọi Try(3) -> gán x[3]=0-> (i==n) nên in ra cấu hình 010.

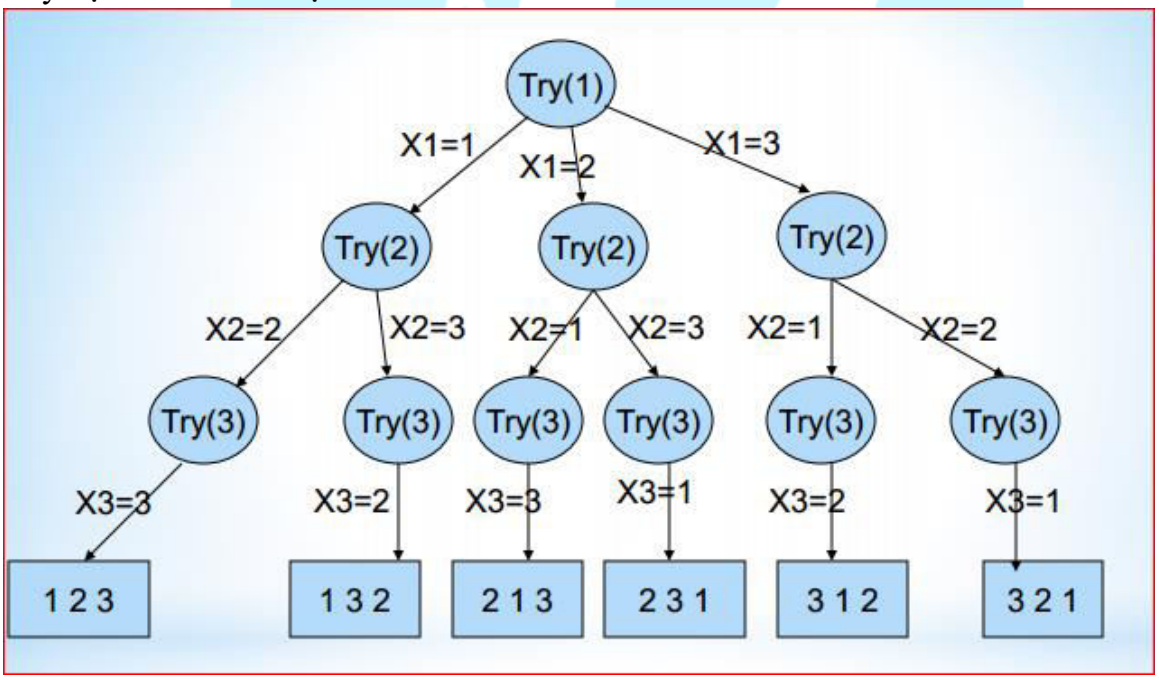
Tương tự với các cấu hình còn lại.

### b. Liệt kê các hoán vị của n phần tử 1, 2, …, n

**Lời giải**. Mỗi hoán vị X = (x1, x2,..,xn) là bộ có tính đến *thứ tự* của 1, 2, .., n. Mỗi xi∈X có N *lựa chọn*. Khi xi = j được lựa chọn thì giá trị này sẽ **không được** chấp thuận cho các thành phần còn lại. Để **ghi nhận** điều này, ta sử dụng mảng chuaxet[] gồm n phần tử. Nếu chuaxet[i] = true điều đó có nghĩa giá trị i **được** chấp thuận và chuaxet[i] = false tương ứng với giá trị i **không được** phép sử dụng. Dựa vào mô hình chung, ta có thể viết thủ tục Try(i) giải quyết bài toán như sau:

```
5   int n,x[100];
6   bool chuaxet[100];
7   void xuat(){
8       for (int i=1;i<=n;i++) cout<<x[i];
9       cout<<endl;
10  }
11  void Try(int i){
12      for (int j=1;j<=n;j++){
13          if (chuaxet[j]){
14              x[i]=j;
15              chuaxet[j]=false;
16              if (i==n) xuat();
17              else Try(i+1);
18              chuaxet[j]=true;
19          }
20      }
21  }
```

Chú ý ban đầu ta phải **khởi tạo** các phần tử của mảng chuaxet bằng true.
Cây liệt kê các hoán vị với n = 3:




Tương tự như với cây liệt kê các dãy nhị phân, nhưng ở đây để ý rằng phần tử j nào chuaxet[j] =true mới được chấp thuận cho các phần tử x[i] tiếp sau của cấu hình.
Ví dụ với 2 cấu hình đầu tiên:

- Gọi Try(1): do chuaxet[1] = true, gán x[1] = 1; gán lại chuaxet[1] = fasle; -> đi sâu vào Try(2): do chuaxet[1] = false nên không thể dùng, sang j = 2 do chuaxet[2] = true, gán x[2] = 2; gán lại chuaxet[2] = fasle; -> đi sâu vào Try(3): do chuaxet[1] = false và chuaxet[2] = false nên không thể dùng, sang j = 3 do chuaxet[3] = true, gán x[3] = 3, gán lại chuaxet[3] = fasle; -> (i==n) nên xuất cấu hình 123

- Bây giờ gán lại chuaxet[3] = true để bỏ việc ghi nhận 3 đã chọn. Quay lui lại Try(2): thực hiện câ lệnh gán lại chuaxet[2] = true và thực hiện vòng for còn dở, đến j = 3 do chuaxet[3] = true, gán x[2] = 3, gán lại chuaxet[3] = fasle; -> đi vào Try(3): chỉ còn chuaxet[2] = true nên gán x[3] = 2, gán lại chuaxet[2] = fasle; -> (i==n) nên xuất cấu hình 132.
Tương tự với các cấu hình còn lại.

## III. Một số bài toán đặc trưng sử dụng thuật toán Đệ quy – Quay lui

### 1. Bài toán tìm đường đi trên ma trận

Cho ma trận a: mxn (1<=m,n<=20). Mỗi phần tử trên ma trận chỉ nhận 2 giá trị 0 hoặc 1 với ý nghĩa: a[i][j] = 1 nếu *có thể* đi qua ô (i,j) trên ma trận, a[i][j] = 0 nếu tại ô (i,j) có chướng ngại vật *không thể* đi qua. Cho trước m,n và giá trị các phần tử của a. Hãy **liệt kê** các **đường đi** từ ô (1,1) đến ô (m,n), với mỗi lần đi chỉ được di chuyển **sang phải** hoặc **xuống dưới**. (a[1][1] và a[m][n] luôn bằng 1). Mỗi đường đi thỏa mãn ghi ra trên **một dòng** một xâu chỉ gồm 2 kí tự "R" (sang phải) và "D" (xuống dưới).

**VD:**

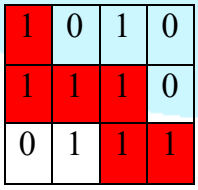
**Input:**

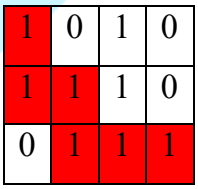
3 4

1 0 1 0

1 1 1 0 => hoặc

0 1 1 1





**Output:**

DRDRR

DRRDR

**Lời giải:**

- Ta dễ dàng nhận thấy với *bất kì* cách đi nào thì *số lần* đi từ ô (1,1) đến ô (m,n) cũng là **m+n-2** lần đi.

- Có 2 *cách đi* tại mỗi lần đi là sang phải và xuống dưới. Do vậy ta nghĩ ra một cách có thể liệt kê được *tất cả* các đường đi có thể có từ ô (1,1) đến ô (m,n). Ta sẽ dùng sinh các dãy nhị phân độ dài m+n-2. Với mỗi dãy nhị phân x sinh ra: nếu **x[i] = 0** thì đi xuống, nếu **x[i] = 1** thì sang phải.

```
30 void Try(int i){
31     for (int j=0;j<=1;j++){
32         x[i]=j;
33         if (i==m+n-2) kt();
34         else Try(i+1);
35     }
36 }
```

- Công việc của ta sau khi sinh được 1 dãy nhị phân là kiểm tra xem dãy đó có là 1 đường đi *thỏa mãn* không. Cách kiểm tra là dùng một biến logic ***ok*** khởi tạo bằng **true**. Khởi tạo hàng cột bắt đầu h = 1, c= 1. Chạy vòng for từ 1 đến m+n-2 thử đi theo đường x tạo ra. Nếu x[i] = 0 thì xuống dưới: h=h+1; x[i] = 1 thì sang phải: c = c+1. Nếu trên đường đi gặp a[h][c] = 0 hoặc đi ra khỏi ma trận: **h>m** hoặc **c>n**, thì đều **gán lại** ok = false và **thoát**. Cuối cùng nếu ok vẫn bằng true nghĩa là đường đi *thỏa mãn*. Ta chỉ cần in ra đường đi đó: x[i] = 0: in ra "D", ngược lại in ra "R".

```
12 void kt(){
13     bool ok = true;
14     int h = 1,c = 1;
15     for (int i=1;i<=m+n-2;i++){
16         if (x[i]==0) h = h+1;
17         else c = c+1;
18         if (a[h][c]==0 || h>m || c>n){
19             ok = false;
20             break;
21         }
22     }
23     if (ok){
24         for (int i=1;i<=m+n-2;i++)
25             if (x[i]==0) cout<<"D";
26             else cout<<"R";
27         cout<<endl;
28     }
29 }
```

## 2. Bài toán người du lịch

**Link đề bài:** http://www.spoj.com/PTIT/problems/BCTSP/

Cho n thành phố đánh số từ 1 đến n và các tuyến đường giao thông hai chiều giữa chúng, mạng lưới giao thông này được cho bởi mảng C[1…n, 1…n] ở đây C[i][j] = C[j][i] là chi phí đi đoạn đường trực tiếp từ thành phố I đến thành phố j.

Một người du lịch xuất phát từ thành phố 1, muốn đi thăm tất cả các thành phố còn lại mỗi thành phố đúng 1 lần và cuối cùng quay lại thành phố 1. Hãy chỉ ra chi phí ít nhất mà người đó phải bỏ ra.

**Input**

Dòng đầu tiên là số nguyên n – số thành phố (n <= 15)

n dòng sau, mỗi dòng chứa n số nguyên thể hiện cho mảng 2 chiều C.

**Output**

Chi phí mà người đó phải bỏ ra.

**Example**

```
Input:
4
0 20 35 10
20 0 90 50
35 90 0 12
10 50 12 0

Output:
117
```

**Lời giải:**

- Mỗi hành trình từ thành phố 1 đi qua tất cả các thành phố, mỗi thành phố đúng **1 lần** là một *hoán vị* của 1, 2,…, n, với **x[1] = 1**. Ta nghĩ ngay đến việc sinh các *hoán vị* của 1, 2,…, n với x[1] = 1, mỗi khi sinh được một hoán vị thì cộng lại toàn bộ chi phí của cách đi đó và so sánh với chi phí *nhỏ nhất* hiện tại, nếu nhỏ hơn thì sẽ *cập nhật* lại. Chú ý khởi tạo x[1] = 1, chi phí nhỏ nhất ban đầu MIN là 1 số *đủ lớn* để có thể cập nhật ngay lần đầu tiên, và công việc của chúng ta là gọi Try(2).

```
11 void Try(int i){
12     for (int j=2; j<=n; j++)
13         if (chuaxet[j]){
14             x[i]=j;
15             chuaxet[j]=false;
16             if (i==n) kt();
17             else Try(i+1);
18             chuaxet[j]=true;
19         }
20 }
```

- Bài này chỉ cần viết thêm vào bài liệt kê hoán vị hàm ***kt()*** tính chi phí tổng của mỗi cách đi như sau:

```
5  void kt(){
6      long chiphi = c[x[n]][x[1]];
7      for (int i=2;i<=n;i++)
8          chiphi+=c[x[i-1]][x[i]];
9      if (chiphi<MIN) MIN = chiphi;
10 }
```

Ví dụ như n = 3 và hoán vị của ta là 123, thì chi phí là c[1][2]+c[2][3]+c[3][1]. Sau khi đã thử toàn bộ các hoán vị thì chỉ cần in ra MIN chính là chi phi nhỏ nhất.

## 3. Bài toán xếp hậu

**Link bài**: http://www.spoj.com/PTIT/problems/BCQUEEN/

**Đề bài** : Cho một bàn cờ vua có kích thước n * n, ta biết ràng quân hậu có thể di chuyển theo chiều ngang, dọc, chéo. Vấn đề đặt ra rằng, có n quân hậu, bạn cần đếm số cách đặt n quân hậu này lên bàn cờ sao cho với 2 quân hậu bất kì, chúng không "ăn" nhau.

**Input**: Số nguyên dương n duy nhất (n<=10)

**Output**: Các cách sắp xếp quân hậu trên bàn cờ thỏa mãn yêu cầu đề bài, với mỗi bàn cờ được coi như một ma trận vuông cấp n mà các phần tử đều bằng 0, tại vị trí đặt quân hậu thì phần tử đó bằng 1. Dòng cuối cùng in ra số cách sắp xếp thỏa mãn duy nhất. Trường hợp không tìm được cách sắp xếp nào thì in ra số 0.

VD:

| Test 1 | Test 2 |
|---|---|

| | |
|---|---|
| Input:<br>3<br>Output:<br>0 | Input:<br>4<br>Output:<br>0 1 0 0<br>0 0 0 1<br>1 0 0 0<br>0 0 1 0<br><br>0 1 0 0<br>0 0 0 1<br>1 0 0 0<br>0 0 1 0<br><br>2 |

**Lời giải:** Ta sẽ coi bàn cờ là mảng 2 chiều a có n hàng n cột, chỉ số bắt đầu từ 1

+ Đường *chéo chính* là đường chéo nối giữa 2 ô a[1][1] và a[n][n]

+ Đường *chéo phụ* là đường chéo nối giữa 2 ô a[n][1] và a[1][n]

Bài toán sử dụng phương pháp đệ quy quay lui + kĩ thuật đánh dấu:

+/ khi một quân hậu được đặt vào ô [i][j] thì các quân hậu sau sẽ *không thể* đặt được vào hàng i, cột j và hai đường chéo mà nó có thể ăn được.

+/ Do 1<= chỉ số cột <=n nên ta sẽ dùng một mảng một chiều **cot[]** *đủ lớn* để đánh dấu các cột đã đặt quân hậu. Với ô tại hàng h, cột c, nếu cot[c] = 0 nghĩa là **có thể** đặt thêm quân hậu, ngược lại nếu cot[c] = 1 nghĩa là đã có quân hậu đặt ở hàng c rồi và **không thể** đặt thêm.

+/ các ô tọa độ [h][c] nằm trên đường chéo đi qua ô đặt quân hậu [i][j] *song song hoặc trùng* với đường *chéo phụ* có tổng h+c = i + j = **không đổi**. Do 2<=h+c<=2*n nên ta sẽ dùng mảng 1 chiều **cheo1[]** *đủ lớn* để đánh dấu. Với ô tại hàng h, cột c, nếu cheo1[h+c] = 0 thì **đặt được** quân hậu, ngược lại nếu cheo1[h+c] = 1 nghĩa là **không thể** đặt thêm hậu tại ô [h][c].

+/ các ô tọa độ [h][c] nằm trên đường chéo đi qua ô đặt quân hậu [i][j] *song song hoặc trùng* với đường *chéo chính* có hiệu h-c = i - j = **không đổi**. Do 1-n<=h-c<=n-1 nên ta sẽ ***không thể*** dùng mảng để đánh dấu lại hiệu đấy do chỉ số **bị âm**. Do vậy ta **cộng** thêm 1 lượng là n, khi đó 1<=h-c+n<=2*n-1, lúc này ta dùng mảng 1 chiều

**cheo2[]** *đủ lớn* để đánh dấu. Với ô tại hàng h, cột c, nếu cheo2[h-c+n] = 0 thì **đặt được** quân hậu, ngược lại nếu cheo2[h-c+n] = 1 nghĩa là **không thể** đặt thêm hậu tại ô [h][c].

=> sử dụng đệ quy quay lui tìm cách đặt quân hậu theo từng **hàng** rồi đánh dấu lại **cột** và hai **đường chéo**, nếu tìm được đến **hàng cuối** thì in ra kết quả và tăng biến **đếm**, nếu không tìm được đến hàng cuối thì *quay lui* lại để đặt lại quân hậu phía trước.

***Khởi tạo***:

```
#include<iostream>
using namespace std;
int n,cot[100]={0},cheo1[200]={0},cheo2[200]={0},d=0,a[100][100]={0};
```

***Hàm in kết quả***:

```
void in(){
    for(int i=1;i<=n;i++){
        for(int j=1;j<=n;j++) cout<<a[i][j]<<" ";
        cout<<endl;
    }
    cout<<endl;
}
```

***Hàm đệ quy quay lui***:

```
void timhau(int i){
    if(i==n+1){
        in();
        d++;
        return;
    }
    for(int j=1;j<=n;j++){
        if(cot[j]==0 && cheo1[i+j]==0 && cheo2[i-j+n]==0){
            cot[j]=1;
            cheo1[i+j]=1;
            cheo2[i-j+n]=1;
            a[i][j]=1;
            timhau(i+1);
            cot[j]=0;
            cheo1[i+j]=0;
            cheo2[i-j+n]=0;
            a[i][j]=0;
        }
    }
}
```

***Hàm main***:

```
int main(){
    cin>>n;
    timhau(1);
    cout<<d;
}
```

## 4. Bài toán quân mã đi tuần

**Link bài** : http://www.spoj.com/PTIT/problems/BCKNIGHT/

**Đề bài** : Cho một bàn cờ vua kích thước n * n, ta đặt sẵn một quân mã tại ô (x, y), nhiệm vụ của chúng ta là tìm cách đi cho quân mã sao cho quân mã sẽ đi qua tất cả các ô trên bàn cờ và mỗi ô đi tới chính xác một lần duy nhất.

**Input**: Gồm 3 số nguyên n, x, y lần lượt là kích thước bàn cờ, và vị trí đặt quân mã (1 <= n <= 8, 1<=x, y <= n)

**Output**: Bàn cờ kích thước n * n, mỗi ô (x, y) ghi một số p với ý nghĩa khi quân mã đi tới ô (x, y) thì số bước đã đi là p. Ô (x, y) ta ghi là bước 1. Dữ liệu đảm bảo bài toán có một đáp án duy nhất, và thời gian thực không quá 1s nếu đúng thuật toán quay lui cơ bản.

**VD:**

**Input:**

6 2 3

**Output:**

36 17 6 29 8 11

19 30 1 10 5 28

16 35 18 7 12 9

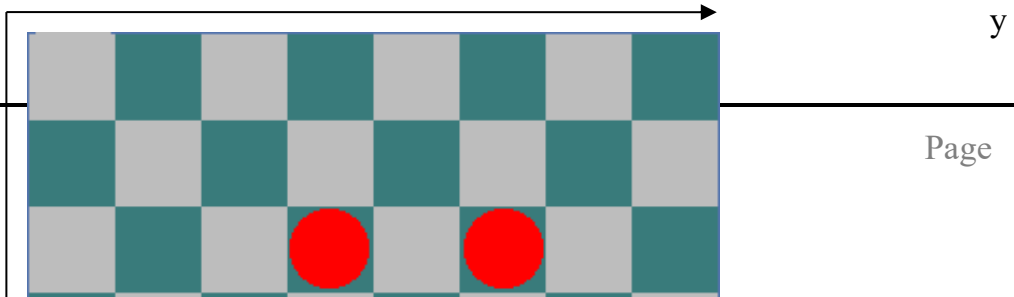
23 20 31 2 27 4

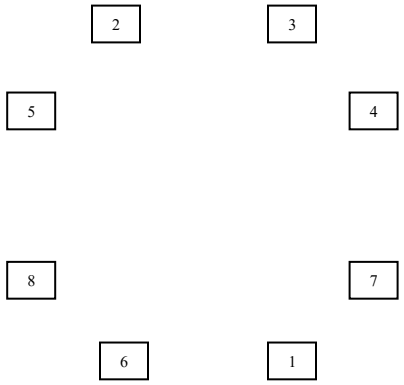
34 15 22 25 32 13

21 24 33 14 3 26

- Bài toán sử dụng đệ quy quay lui bằng cách từ ô hiện tại *thử* đi quân mã vào 1 trong 8 ô nó **có thể đến**, nếu đi được thì đánh dấu ô đấy lại và từ ô đấy thử đi tiếp đến 8 ô có thể đến tiếp theo. Nếu đi được **n*n** ô thì in ra đáp án.

- Ta sử dụng hai **mảng hằng** để biểu diễn cách di chuyển quân mã đến 8 ô nó có thể đến.






x

- Giả sử quân mã đang có tọa độ **(x,y)**, khi đó nó có thể di chuyển đến các ô đỏ từ 1 đến 8 như hình vẽ với tọa độ các ô đỏ lần lượt là:

| **Vị trí** | **Tọa độ** |
|---|---|
| 1 | (x+2,y+1) |
| 2 | (x-2,y-1) |
| 3 | (x-2,y+1) |
| 4 | (x-1,y+2) |
| 5 | (x-1,y-2) |
| 6 | (x+2,y-1) |
| 7 | (x+1,y+2) |
| 8 | (x+1,y-2) |

-Từ đó ta sẽ sử dụng một mảng hằng a[8]={2,-2,-2,-1,-1,2,1,1} để biểu diễn theo hàng x, và một mảng hằng b[8]= {1,-1,1,2,-2,-1,2,-2} để biểu diễn theo cột y.

-Khởi tạo mảng 2 chiều h[][] với tất cả phần tử **bằng 0**, vị trí đặt quân mã là **h[x][y]=1**, ta đệ quy quay lui bắt đầu với giá trị số bước đi bắt đầu từ 2. Thủ tục **Try(i,x,y)** có ý nghĩa quân mã đang ở ô **(x,y)** với số bước đi là **i-1**và nếu đi được đến 1 ô **(x',y')** khác thì số bước đi sẽ là **i**, tiếp tục đi vào **Try(i+1,x',y')**…

-Trong hàm đệ quy sử dụng một vòng lặp với biến j chạy từ 0 đến 7 để tìm vị trí tiếp theo của quân mã (VD: j=0 thì ứng với tọa độ của mã là (x+a[0],y+b[0])). Nếu tại vị trí **(x, y)** mà mã không thể di chuyển đến (x+a[0],y+b[0]) thì j++.

-Giả sử, mã có thể đi được vào ô (x+a[0],y+b[0]). Đặt u=x+a[0], v=y+b[0]. Điều kiện để chương trình đệ quy `void Try(int i,int x,int y)` đi sâu vào tìm được vị trí tiếp theo của quân mã là 1 <= u,v <= n và vị trí đó quân mã chưa đi qua (tức là h[u][v]=0). Nếu thỏa mãn thì gán h[u][v]=i.

- Nếu i < n*n thì đi sâu tìm `void Try(int i+1,int x,int y)`
- Ngược lại, nghĩa là khi đã tìm đủ tất cả các vị trí trên bàn cờ thì in ra ma trận.

***Khởi tạo:***

```
#include<iostream>
using namespace std;
int h[11][11]={0};
int n;
int a[8] ={2,2,1,1,-1,-1,-2,-2};
int b[8] ={1,-1,2,-2,2,-2,1,-1};
```

***Hàm in kết quả:***

```
void in(){
    for(int j=1;j<=n;j++){
        for(int l=1;l<=n;l++) cout<<h[j][l]<<" ";
        cout<<endl;
    }
}
```

***Hàm đệ quy quay lui:***

```
void Try(int i,int x,int y){
    int u, v;
    for(int k=0;k<=7;k++){
        u=x+a[k]; v=y+b[k];
        if(1<=u && u<=n && 1<=v && v<=n && h[u][v]==0){
            h[u][v]=i;
            if(i<n*n) {
                Try(i+1,u,v);
                h[u][v]=0;
            }
            else in();
        }
    }
}
```

***Hàm main:***

```
int main(){
    int x,y;
    cin>>n>>x>>y;
    h[x][y]=1;
    Try(2,x,y);
}
```

## IV. Kỹ thuật nhánh cận

### 1. Bài toán tối ưu và sự bùng nổ tổ hợp

#### a. Bài toán tối ưu

- Một trong những bài toán đặt ra trong thực tế là việc tìm ra một nghiệm thỏa mãn một số điều kiện nào đó, và nghiệm đó là tốt nhất theo một chỉ tiêu cụ thể. Trong nhiều trường hợp, ta chưa thể xây dựng một thuật toán thực sự hữu hiệu để giải bài toán, mà vẫn phải dựa trên mô hình liệt kê toàn bộ các cấu hình thỏa mãn, đánh giá và tìm ra cấu hình tốt nhất.

#### b. Sự bùng nổ tổ hợp

- Ta thấy ở thuật toán quay lui, chúng ta sẽ liệt kê toàn bộ cấu hình có thể có. Ví dụ như sinh dãy nhị phân độ dài n, với n = 30, số cấu hình có thể là 2^30 tầm khoảng hơn 1 tỉ cấu hình. Sinh hoán vị với n = 13, số cấu hình là 13! tầm khoảng hơn 6 tỉ cấu hình. Nếu thử đánh giá toàn bộ các cấu hình có thể sinh ra, thì thời gian chờ đợi là vô cùng lớn đối với tốc độ của máy tính hiện nay!

Do vậy cần phải có phương pháp đánh giá ngay tại thời điểm quyết định chọn x[i], nếu chọn x[i] là thừa, nghĩa là các x[i+1],x[i+2],.. dù có chọn thế nào thì cấu hình hiện tại vẫn không khả quan, thì sẽ không chọn x[i] nữa.

- Khi đó, một vấn đề đặt ra trong quá trình liệt kê lời giải là ta cần tận dụng những thông tin đã tìm được để loại bỏ sớm những phương án chắc chắn không phải tối ưu. Kỹ thuật đó gọi là kỹ thuật đánh giá nhánh cận trong tiến trình quay lui.

### 2. Mô hình kỹ thuật nhánh cận

- Dựa trên mô hình thuật toán quay lui, ta xây dựng mô hình sau:

```
void tim (int i)
{
    for( Các giá trị j có thể gán cho x[i])
    {
        x[i]=j;
        if( Việc thử trên vẫn còn hi vọng tìm ra cấu hình tốt hơn BESTCONFIG)
        {
            if( x[i] là phần tử cuối cùng)
                Cập nhật lại BEST CONFIG
            else
            {
                Ghi nhận việc thử x[i]=j nếu cần
                tim(i+1);
                Bỏ ghi nhận việc thử x[i]=j nếu cần
            }
        }
    }
}
```

- Kỹ thuật nhánh cận thêm vào cho thuật toán quay lui khả năng đánh giá theo từng bước, nếu tại bước thứ i, giá trị thử gán cho x[i] không có hi vọng tìm thấy cấu hình tốt hơn BESTCONFIG thì thử ngay giá trị khác mà không cần phải gọi đệ quy tìm tiếp hay ghi nhận kết quả. Nghiệm của bài toán sẽ được làm tốt dần, vì khi tìm ra một cấu hình mới tốt hơn thì BESTCONFIG sẽ được cập nhật lại bằng cấu hình đó.

- Một bài toán có thể có nhiều cách đánh giá nhánh cận khác nhau. Việc tìm được cận càng chặt sẽ giảm bớt càng nhiều chi phí tính toán. Tuy nhiên, việc tìm ra cách đánh giá nhánh cận hiệu quả cho nhiều bài toán là không dễ dàng.

## 3. Một số bài toán sử dụng kỹ thuật nhánh cận

### a. Bài toán người du lịch

**Link đề bài:** http://www.spoj.com/PTIT/problems/BCTSP/

Cho n thành phố đánh số từ 1 đến n và các tuyến đường giao thông hai chiều giữa chúng, mạng lưới giao thông này được cho bởi mảng C[1…n, 1…n] ở đây C[i][j] = C[j][i] là chi phí đi đoạn đường trực tiếp từ thành phố I đến thành phố j.

Một người du lịch xuất phát từ thành phố 1, muốn đi thăm tất cả các thành phố còn lại mỗi thành phố đúng 1 lần và cuối cùng quay lại thành phố 1. Hãy chỉ ra chi phí ít nhất mà người đó phải bỏ ra.

Input

Dòng đầu tiên là số nguyên n – số thành phố (n <= 15)

n dòng sau, mỗi dòng chứa n số nguyên thể hiện cho mảng 2 chiều C.

Output

Chi phí mà người đó phải bỏ ra.

**Example**

```
Input:
4
0 20 35 10
20 0 90 50
35 90 0 12
10 50 12 0

Output:
117
```

Quay trở lại với bài toán người du lịch ta đã tiếp cận trong thuật toán quay lui. Nếu bạn nào đã làm và nộp thử bài này theo cách đã làm trong quay lui là liệt kê toàn bộ các cấu hình và cập nhật lại chi phí nhỏ nhất sau khi sinh ra 1 cấu hình thì sẽ thấy: kết quả **chính xác** nhưng bị **quá thời gian**. Lý do là n<=15, với n = 15 thì số cấu hình là 15!, **rất lớn** nên không thể chạy trong thời gian **cho phép**. Chính vì thế, ta sẽ sử dụng kỹ thuật đánh giá *nhánh cận* để làm bài này trong thời gian cho phép.

**Lời giải:**

Gọi *c*min = min {*c*[i, j], *i , j = 1, 2,.., n, i≠j* } là giá trị *nhỏ nhất* của ma trận chi phí.
Phương pháp đánh giá cận dưới của mỗi bài toán bộ phận cấp k được tiến hành như sau. Giả sử ta đang có hành trình bộ phận qua k thành phố:
T 1 →Tu2 →.. →Tuk (T1 =1).
Khi đó, chi phí của phương án bộ phận cấp k là:
S = c[1, u2] + c[u2,u3] +…+ c[uk-1, uk].
Để phát triển hành trình bộ phận này thành hành trình đầy đủ, ta cần phải qua n-k thành phố nữa rồi quay trở về thành phố số 1. Như vậy, ta cần phải qua n-k+1 đoạn đường nữa. Vì mỗi đoạn đường đều có chi phí *không nhỏ hơn* cmin, nên cận dưới của phương án bộ phận có thể được xác định = S + (n-k+1).cmin. Chi phí của *phương án hiện tại* luôn >= S + (n-k+1).cmin.

Trong thủ tục **Try(int i),** ta sẽ tính S tại mỗi phương án bộ phận, và nếu S + (n-i+1).cmin *nhỏ hơn* chi phí nhỏ nhất là **MIN** hiện tại thì mới đệ quy tính tiếp, ngược lại **thử ngay** phương án khác do có thử tiếp thì chi phí cũng **>=MIN** mà thôi.

```
16 void Try(int i){
17     for (int j = 2; j<= n; j++)
18         if (chuaxet[j]){
19             x[i] = j;
20             chuaxet[j] = 0;
21             S+=c[x[i-1]][j];
22             if (i == n){
23                 if (S+c[j][1]<MIN) MIN = S+c[j][1];
24             }
25             else if (S+(n-i+1)*cmin<MIN) Try(i+1);
26             S-=c[x[i-1]][j];
27             chuaxet[j] = 1;
28         }
29 }
```

## b. Bài toán cái túi

Trong siêu thị có n gói hàng (n <= 20), gói hàng thứ i có trọng lượng là $W_i$ <= 100 và giá trị $V_i$ <= 100. Một Tên trộm đột nhập vào siêu thị, sức của tên trộm không thể mang được trọng lượng vượt quá M (M <= 100). Hỏi tên trộm sẽ lấy đi những gói hàng nào để được tổng giá trị lớn nhất.

Input

Dòng đầu tiên gồm 2 số nguyên n và M (n <= 20, M <= 100)

Trên n dòng tiếp theo, mỗi dòng chứa 2 số nguyên $W_i$ và $V_i$ ($W_i$, $V_i$ <= 100) lần lượt là trọng lượng và giá trị của gói hàng thứ i.

Output

Giá trị lớn nhất tên trộm lấy được.

**Example**

```
Input:
3 4

1 4
```

```
2 5

3 6

Output:
10
```

**Lời giải:**

- Mỗi phương án của bài toán là một xâu nhị phân có độ dài *n*. Trong đó, *x*i =1 tương ứng với đồ vật i được *đưa vào* túi, *x*i =0 tương ứng với đồ vật i *không được đưa* vào túi. Tập các xâu nhị phân *X* =(*x*1,..,*x*n) còn phải *thỏa mãn* điều kiện tổng trọng lượng *không vượt* quá M.

- Ta dễ dàng giải quyết bài toán này bằng cách liệt kê toàn bộ các cấu hình có thể và cập nhật kết quả. Tuy nhiên thời gian **không cho phép**. Do vậy cũng cần sử dụng *nhánh cận* trong bài toán này. Có một cách đánh giá nhánh cận khá *hay và hiệu quả* đối với bài toán này như sau:

**Bước 1:** Sắp xếp các đồ vật thỏa mãn: v1/w1>=v2/w2>=…>=vn/wn

```
11 void sort(){
12     for (int i = 1; i<=n-1; i++)
13         for (int j = i+1; j<=n; j++)
14             if ((double) v[i]/w[i]<(double) v[j]/w[j]){
15                 swap(v[i],v[j]);
16                 swap(w[i],w[j]);
17             }
18 }
```

**Bước 2**: Lặp trên các bài toán bộ phận cấp i=1,2,..,n:

- Giá trị của i đồ vật trong túi: V = $\sum_{j=1}^{i} xj.vj$
- Trọng lượng còn lại của túi: W = M - $\sum_{j=1}^{i} xj.vj$
- Cận trên của phương án bộ phận cấp i: V+W. $\frac{v[i+1]}{w[i+1]}$

Tức là giá trị lớn nhất của phương án hiện tại khi trở thành phương án hoàn chỉnh luôn <=cận trên. Trong thủ tục **Try(int i)** nếu cận trên của phương án hiện tại **>MAX** thì mới

đệ quy tính tiếp, nếu không thử ngay phương án khác vì có thử thì cũng sẽ thu được kết quả **<=MAX** mà thôi.

```
19  void Try(int i){
20      for (int j = 1; j>=0; j--){
21              x[i] = j;
22              V += v[i]*x[i];
23              W -= w[i]*x[i];
24              if (i == n){
25                  if (W>= 0 & V>MAX) MAX = V;
26              }
27              else if ((double)V+(double)(v[i+1]*W)/w[i+1]>(double) MAX) Try(i+1);
28              V -= v[i]*x[i];
29              W += w[i]*x[i];
30          }
31  }
```

**Bước 3:** Trả lại kết quả phương án tối ưu tìm được.

Chú ý ban đầu ta khởi tạo giá trị lớn nhất thu được **MAX = 0;** (chưa có đồ vật nào). Sau khi xong thủ tục **Try(int i)** thì **MAX** đã được cập nhật chính là giá trị *lớn nhất* của các đồ vật bỏ vào túi trong phương án *tối ưu.*

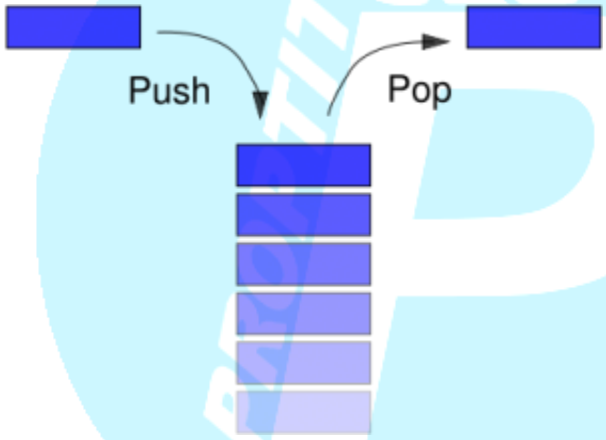
# Bài 4: Cấu trúc dữ liệu STACK – QUEUE

## I. STACK – Ngăn xếp

### 1. Khái niệm:

Một ngăn xếp là một cấu trúc dữ liệu dạng thùng chứa (container) của các phần tử (thường gọi là các nút (node)) và có hai phép toán cơ bản: push and pop. Push bổ sung một phần tử vào đỉnh (top) của ngăn xếp, nghĩa là sau các phần tử đã có trong ngăn xếp. Pop giải phóng và trả về phần tử đang đứng ở đỉnh của ngăn xếp. Trong stack, các đối tượng có thể được thêm vào stack bất kỳ lúc nào nhưng chỉ có đối tượng thêm vào sau cùng mới được phép lấy ra khỏi stack.

***VD thực tế :*** có thể coi stack như 1 băng đạn , mỗi viên đạn là 1 phần tử. chỉ có thể lắp đạn vào đầu băng đạn (push ) và mỗi lần bắn sẽ bắn viên đạn trên cùng băng đạn ( pop). Hay còn gọi là **LIFO** (Last In First Out) vào trước ra trước.




### 2. Khai báo:

#include<stack> // thư viện

Ví dụ khai báo stack kiểu int

stack<int> s; (nếu kiểu kí tự thì khia báo stack<char>s)

s.size() : trả về kích thước hiện tại của stack (số phần tử trong stack).

s.empty() : true (1) stack nếu rỗng, và (0) ngược lại.

s.push(x) : đẩy phần tử x vào **đỉnh** stack.

s.pop() : loại bỏ phần tử ở **đỉnh** của stack.

s.top() : truy cập tới phần tử ở **đỉnh** stack.

Tất cả hàm trên đều có Độ phức tạp O(1);

### 3. Ví dụ

Ta có 1 stack rỗng rồi thực hiện các thao tác như bên dưới

```
#include <iostream>
#include <stack>

using namespace std;
stack <int> s;                    // khai bao stack kieu int

main() {
    for (int i=1;i<=5;i++) s.push(i);    // s={1,2,3,4,5}
    s.push(100);                         // s={1,2,3,4,5,100}
    cout << s.top() << endl;             // In ra 100
    s.pop();                             // s={1,2,3,4,5}
    cout << s.empty() << endl;           // In ra 0
    cout << s.size() << endl;            // In ra 5
}
```

### 4. Ứng dụng

-Đảo ngược xâu: nhập từng kí tự xâu cho vào stack, sau đó lấy từng kí tự ra ta sẽ được 1 xâu đảo ngược.

- Chuyển hệ cơ số 10 sang cơ số 2: thực hiện liên tiếp phép chia dư n cho 2 rồi , n=n/2 và push kq phép chia dư vào stack, sau khi chia xong ta lấy các phần tử trong stack ra.
- Tính biểu thức đại số, chuyển biểu thức đại số dạng trung sang hậu tố.
- Khử đệ quy (trong duyệt đồ thị DFS)

### 5. Bài tập ứng dụng:

Cho các đoạn văn chứa các dấu ngoặc, có thể là ngoặc đơn đơn ( “()” ) hoặc ngoặc vuông ( “[]” ). Một đoạn văn đúng là đoạn mà với mỗi dấu mở ngoặc thì sẽ có dấu đóng ngoặc tương ứng và đúng thứ tự. Nhiệm vụ của bạn kiểm tra xem đoạn văn có đúng hay không.

| Input | Output |
|---|---|
| Gồm nhiều bộ test, mỗi bộ test trên một dòng chứa đoạn văn cần kiểm tra có thể bao gồm: các kí tự trong bảng chữ cái tiếng Anh, dấu cách, và dấu ngoặc (ngoặc đơn hoặc ngoặc vuông). Kết thúc | Với mỗi bộ test, xuất ra trên một dòng “yes” nếu đoạn văn đúng, |

| | |
|---|---|
| mỗi bộ test là một dấu chấm. Mỗi dòng có không quá 100 kí tự.<br>Dữ liệu kết thúc bởi dòng chứa duy nhất một dấu chấm. | ngược lại in ra “no” |

**Input:**

```
So when I die (the [first] I will see in (heaven) is a score list ).
[ first in ] ( first out ).
Half Moon tonight (At least it is better than no Moon at all].
A rope may form )( a trail in a maze.
Help( I[m being held prisoner in a fortune cookie factory)].
([ ((([( [ ] ) ( ) (( ))] )) ]).
 .
.
```

**Output:**

```
yes
yes
no
no
no
yes
yes
```

***Ý tưởng bài toán***

+ Nếu gặp dấu ‘(‘ hoặc ‘[‘ thì push vào trong stack

+Nếu gặp dấu ‘)’ : nếu đỉnh của stack là ‘(‘ thì pop ra r xét tiếp và ngược lại in ra “no”;

+Nếu gặp dấu ‘]’ : nếu đỉnh của stack là ‘[‘ thì pop ra r xét tiếp và ngược lại in ra “no”;

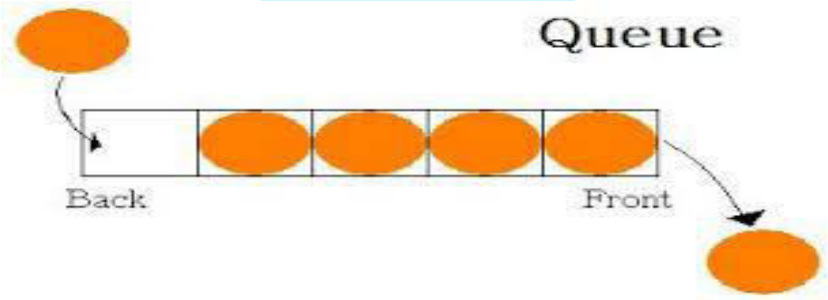
Sau khi xét xong nếu stack rỗng thì in ra “yes”;

## II. QUEUE – Hàng đợi

### 1 .Khái niệm:

**Hàng đợi** ( *queue*) là một cấu trúc dữ liệu dùng để chứa các đối tượng làm việc theo cơ chế **FIFO** (viết tắt từ tiếng Anh: *First In First Out*), nghĩa là "vào trước ra trước", tức là một kiểu danh sách mà việc bổ sung được thực hiện ở cuối danh sách và loại bỏ ở đầu danh sách.

***VD thực tế:*** Việc xếp hàng để giải quyết 1 công việc nào đó, bạn phải xếp hàng vào **cuối** danh sách nếu muốn dc giải quyết công việc, có nghĩa là ai xếp hàng trước sẽ đc giải quyết trước (**đầu** danh sách) .




+ Trong hàng đợi, các đối tượng có thể được thêm vào hàng đợi bất kỳ lúc nào, nhưng chỉ có đối tượng thêm vào đầu tiên mới được phép lấy ra khỏi hàng đợi. Thao tác thêm vào và lấy một đối tượng ra khỏi hàng đợi được gọi lần lượt là "enqueue" và "dequeue". Việc thêm một đối tượng luôn diễn ra ở cuối hàng đợi và một phần tử luôn được lấy ra từ đầu hàng đợi.

+Trong queue, có hai vị trí quan trọng là vị trí đầu danh sách (front), nơi phần tử được lấy ra, và vị trí cuối danh sách (back), nơi phần tử cuối cùng được thêm vào.

### 2. Khai báo:

#include<queue> // thư viện queue
Ví dụ khai báo queue kiểu int
queue <int> s;
s.size() : trả về kích thước hiện tại của queue.
s.empty() : true nếu queue rỗng, và ngược lại
s.push(x) : đẩy x vào **cuối** queue.
s.pop(): loại bỏ phần tử (ở **đầu**).
s.front() : trả về phần tử ở **đầu**
s.back(): trả về phần tử ở **cuối**.

Tất cả hàm trên đều có Độ phức tạp O(1);

### 3. Ví dụ

Ta có 1 queue rỗng rồi thực hiện các thao tác như bên dưới

```
#include <iostream>
#include <queue>
using namespace std;
queue <int> q;
int i;
main() {
for (i=1;i<=5;i++) q.push(i); // q={1,2,3,4,5}
q.push(100); // q={1,2,3,4,5,100}
cout << q.front() << endl; // In ra 1
q.pop(); // q={2,3,4,5,100}
cout << q.back() << endl; // In ra 100
cout << q.empty() << endl; // In ra 0
cout << q.size() << endl; // In ra 5
}
```

### 4. Ứng dụng:

Trong tin học, cấu trúc dữ liệu hàng đợi có nhiều ứng dụng:

+khử đệ quy

+tổ chức lưu vết các quá trình tìm kiếm theo chiều rộng (BFS) và quay lui,vét cạn

+tổ chức quản lý và phân phốitiến trình trong các hệ điều hành, tổ chức bộ đệm bàn phím.

## III. Priority_Queue_Hàng đợi ưu tiên

### 1. Khái niệm:

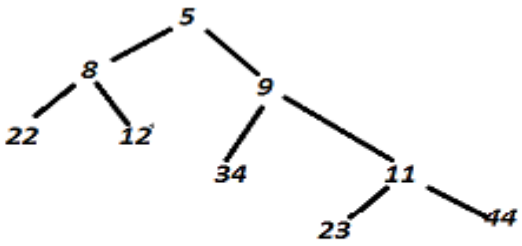
Hàng đợi ưu tiên là một kiểu dữ liệu tập hợp đặc biệt, được xây dựng từ cấu trúc dữ liệu queue nhưng trong đó mỗi phần tử có một độ ưu tiên nhất định.

Đối với hàng đợi ưu tiên,phần tử vào trước chưa chắc sẽ ra trước, nó còn phụ thuộc vào độ ưu tiên của các phần tử trong hàng đợi.

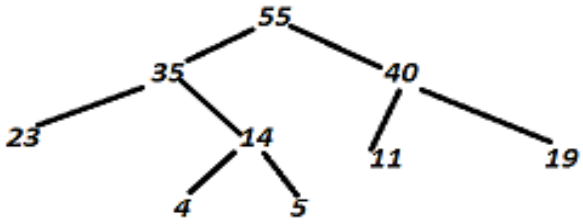
Mục tiêu chính của hàng đợi ưu tiên đó là điều chỉnh để có thể lấy phần tử có độ ưu tiên nhất ra trước. Khi thêm phần tử mới vào thì sắp xếp cho đúng thứ tự ưu tiên lại.

## 2. Cấu trúc Heap:

Heap là một dạng cây nhị phân hoàn chỉnh đặc biệt mà giá trị lưu tại mọi nút có độ ưu tiên cao hơn hay bằng giá trị lưu trong hai nút con của nó.



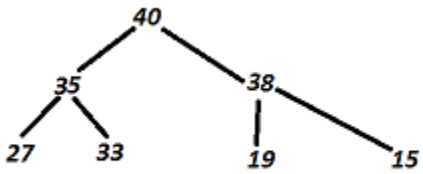

MinHeap




Max Heap

- Cách xây dựng Max Heap:

Giả sử ta đang có một heap như sau:

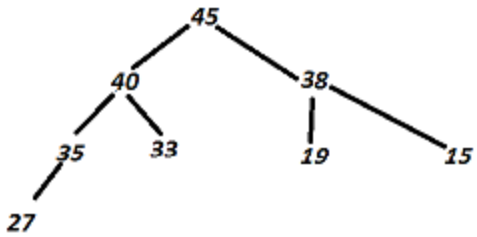



Hình 1

Khi thêm một phần tử x=45 vào heap trên ta thực hiện:

+ Tạo nút mới tại vị trí cuối cùng của heap, nút này có giá trị là 45

+ So sánh giá trị của nút con với giá trị nút cha nếu nhỏ hơn thì đổi chỗ ở đây 45>27 nên 2 nút này đổi giá trị cho nhau. Tiếp tục làm như vậy cho đến khi heap có thuộc tính như ban đầu(phần tử lớn nhất ở đầu các phần tử tiếp theo xếp sau giảm dần). Ta được heap mới như sau:




Hình 2

-Cách xóa đi một phần tử của Max heap:

Với heap như hình 2 phần tử của heap được xóa như sau

+ Xóa nút gốc, di chuyển phần tử cuối cùng lên nút gốc< di chuyển 27 lên vị trí của 45>

+ So sánh giá trị con này với giá trị của cha nếu nhỏ hơn thì đổi chỗ 2 giá trị này. Lặp lại bước này cho đến khi heap có thuộc tính như ban đầu. Ta được heap mới chính là hình 1.

## 3. Cách sử dụng priority queue trong thư viện STL:

- Priority queue được thiết kế để phần tử ở đầu luôn luôn lớn nhất (theo quy ước về độ ưu tiên nào đó) so với các phần tử khác.

- Priority queue giống như một heap mà ở đây là heap max, tức là phần tử có độ ưu tiên lớn nhất có thể được lấy ra và các phần tử khác được chèn vào bất kì.

- Để sử dụng ta cần khai báo thư viện queue.

- Phép toán mặc định khi sử dụng priority queue là phép toán less (nhỏ hơn).

Khai báo: priority_queue <int> x;

- Ngoài ra còn có các phép toán khác :

greater: lớn hơn

equal_to: bằng

not_equal_to: không bằng

greater_equal: lớn hơn bằng

less_equal: nhỏ hơn bằng

- Người dùng cũng có thể tự định nghĩa cách so sánh.

Ta có ví dụ sau: Viết chương trình nhập n. Sau đó nhập mảng gồm n phần tử và in ra kết quả theo thứ tự giảm dần.

```
#include <iostream>
#include <queue>

using namespace std;
priority_queue <int> x;

main(){
    int n,k;
    cin>>n;
    while(n--){
        cin>>k;
        x.push(k);
    }
    while(!x.empty()){
        cout<<x.top();
        x.pop();
    }
}
```

Với sắp xếp giảm dần chỉ cần thay đổi ở “priority_queue<int> x” thành “priority_queue<int ,vector<int> , greater<int> >”

Cũng bài trên nhưng là do người dùng tự định nghĩa như sau:

```cpp
#include <iostream>
#include <queue>

using namespace std;
struct cmp{
    bool operator() (int a,int b){
    return a<b;}
};
priority_queue <int, vector<int>,cmp > x;

main(){
    int n,k;
    cin>>n;
    while(n--){
        cin>>k;
        x.push(k);
    }
    while(!x.empty()){
        cout<<x.top()<<" ";
        x.pop();
    }
}

```

# Bài 5 : Quy hoạch động

## I. Quy hoạch động là gì?

### 1. Đệ quy và quy hoạch động :

- Như các bạn đã biết đến phương pháp đệ quy ở chương trước, phương pháp đệ quy tính toán các bài toán bằng cách giải quyết các bài toán nhỏ hơn thông qua lời gọi hàm đến chính nó. Tuy nhiên phương pháp đệ quy thường sử dụng bộ nhớ lớn và thời gian tính toán khổng lồ. Và quy hoạch động là một kỹ thuật nhằm cải thiện hai nhược điểm lớn trên của đệ quy là sử dụng bộ nhớ lớn và thời gian tính toán rất lớn.

- Quy hoạch động là một phương pháp khó, vì mỗi bài toán tối ưu đều có một đặc thù riêng, nên cách giải quyết hoàn toàn khác nhau đòi hỏi người lập trình phải nắm vững bản chất của quy hoạch động.

### 2. Quy hoạch động là gì ?

- Trạng thái là gì?

Trạng thái là một trường hợp, một bài toán con của 1 bài toán lớn.

- Quy hoạch động :

Quy hoạch động là kĩ thuật được dùng khi có một công thức hoặc một (một vài) trạng thái bắt đầu. Một bài toán được tính bởi các bài toán nhỏ hơn đã được tìm ra từ trước, và kết quả các bài toán sẽ được lưu lại để những lần tính toán tiếp theo nếu cần đến những kết quả đó thì không cần tốn thêm thời gian thực hiện lại những bài toán này nữa.

Nói cách khác, quy hoạch động bắt đầu từ việc giải các bài toán nhỏ, để từ đó từng bước giải quyết bài toán lớn hơn , và cuối cùng là bài toán lớn nhất ( bài toán ban đầu).

***QHĐ có độ phức tạp đa thức nên sẽ chạy nhanh hơn quay lui và duyệt trâu.***

### 3. Ưu nhược điểm :

- Ưu điểm : chương trình chạy nhanh, mang tính tối ưu hóa cao

- Nhược điểm :

+ Sự kết hợp giữa các bài toán con chưa chắc cho ta bài toán lớn.

+ Số lượng các bài toán con cần giải quyết có thể rất lớn.

## II. Hai cách tiếp cận quy hoạch động.

***Bài toán ví dụ :***

Tìm số Fibonacci thứ n: xây dựng hàm tính trực tiếp dãy Fibo thứ n theo định nghĩa toán học với công thức truy hồi : F[k] = F[k-1] + F[k-2] , F[0] = 1 , F[1] =1. Cho n tìm F[n].

Chúng ta sẽ tìm cách giải quyết bài toán này bằng các cách tiếp cận dưới đây.

***Cách giải thông thường khi dùng đệ quy, quay lui :***

Độ phức tạp cho phương pháp này là O(2^n) vì mỗi lần gọi hàm tính F[k] ta phải gọi đến hai bài toán con, và phải đi sâu tính lại các bài toán con đó.

Ví dụ : Fibo(4) = Fibo(3)+Fibo(2) , Fibo(3) = Fibo(2)+Fibo(1) : ở đây Fibo(3) phải thực hiện tính 2 lần và với n bài toán đều phải tính lại 2 lần sẽ khiến độ phức tạp là O(2^n), tuy nhiên quy hoạch động có lưu trữ các bài toán và có thể giải quyết điều này.

```
int Fibo(int n){
    if (n==0 ||n == 1)
        return 1;
    else
        return Fibo(n-1) + Fibo (n-2);
}
```

***Quy hoạch động thường dùng một trong hai cách tiếp cận :***

### 1. Top-down (Từ trên xuống):

Bài toán được chia thành các bài toán con, các bài toán con này được giải và lời giải được ghi nhớ để phòng trường hợp cần dùng lại chúng, và các bài toán con này tiếp tục gọi đến bài toán con của chúng đến khi đủ dữ kiện để lấy ra kết quả tính toán.

***Đây là kỹ thuật đệ quy và lưu trữ được kết hợp với nhau, quy hoạch động giải quyết.***

Áp dụng với bài toán ví dụ : Ta sẽ thực hiện lời gọi tính toán trực tiếp đến bài toán gốc Fibo (n) và bài toán này sẽ được chia nhỏ và sau khi tính được kết quả sẽ được lưu trữ vào mảng, và khi gặp lại bài toán đã được tính toán rồi ta sẽ không cần thực hiện đệ quy đi sâu vào để giải quyết của bài toán con đó nữa.

```
int Fibo(int n){
    if (n==0 ||n == 1) return 1;    // bai toan co so

    if (F[n] != 0 )return F[n]; //da duoc tinh truoc do ->return F[n]

    // ---> Neu Fibo (n) chua duoc tinh truoc do
    F[n] = Fibo(n-1) + Fibo (n-2); // luu tru lai vao mang cho lan sau
    return F[n] ; // tra ra ket qua cho loi thuc hien ham Fibo(n)
}
```

Phân tích độ phức tạp : So với phương pháp đệ quy –quay lui, cách tiếp cận này lưu trữ lại những bài toán đã được tính và khi cần không cần phải đi sâu vào nữa, bài toán Fibo(k) gọi đến hai bài toán nhỏ hơn nên số lượng bài toán tối đa là n +1 bài toán Fibo(0) … Fibo(n) và khi tính được đến đâu ta lưu trữ kết quả mỗi bài toán lại đến đó nên bài toán tương đương với tối đa n+1 lần lưu trữ nên độ phức tạp của thuật toán chỉ là O(n). Và đương nhiên quy hoạch động có thể chạy với n lên đến hàng triệu trong khi đệ quy quay lui chỉ chạy với n rất nhỏ (n < 25)

Từ đó cho ta thấy cách giải quyết của quy hoạch động cho ta độ phức tạp chỉ là số lượng bài toán và có thể nhân thêm với chi phí chuyển bài toán (ở những bài toán khác) ở đây là phép tính trực tiếp nên độ phức tạp là O(n*1) ~ O(n).

## 2. Bottom-up (Từ dưới lên)

Tất cả các bài toán con có thể cần đến đều được giải trước (từ bài toán cơ sở trở đi) , sau đó được dùng để xây dựng lời giải cho các bài toán lớn hơn.

*Cách tiếp cận này tốt hơn về không gian bộ nhớ dùng cho ngăn xếp và số lời gọi hàm. Tuy nhiên, đôi khi việc xác định tất cả các bài toán con cần thiết cho việc giải quyết bài toán cho trước không được trực quan như Top-down.*

Áp dụng với bài toán ví dụ :

```
int Fibo(int n){
    F[0] = 1;   F[1] = 1;  // khoi tao bai toan co so
    for (int i=2; i<=n; i++){
        F[i] = F[i-1] + F[i-2];
        // tinh cac bai toan lon dua vao bai toan nho co san
    }
    return F[n];
}
```

Cách tiếp cận này cho ta thấy rõ độ phức tạp của thuật toán quy hoạch động : một vòng for O(n). Ở đây ta thực hiện chuẩn bị sẵn các bài toán đã tính được : đầu tiên là F[0] và F[1] sau đó sẽ lần lượt tính các F[2] -> F[3] … và cuối cùng sẽ được F[n] là kết quả bài toán. Cách tiếp cận này code sẽ ngắn gọn hơn nhiều nhưng với nhiều bài toán phức tạp hơn thì sẽ không trực quan như Top-down do phải điều chỉnh thứ tự gọi các bài toán cho lần lượt ở vòng for để đúng với quy trình đi từ nhỏ đến lớn. Tuy nhiên các tiếp cận này luôn tốt hơn khi cài đặt chính xác do đây là giải thuật khử đệ quy cho bài toán Top-down.

### 3. Phương pháp giải bài toán quy hoạch động :

Để giải quyết bài toán quy hoạch động ta chỉ cần xác định được hai vấn đề :

Thứ nhất là : Bài toán cơ sở.

Thứ hai là : Công thức truy hồi.

Và để có được công thức truy hồi, ta phải xác định được bài toán với đủ dữ kiện để có thể xây dựng được công thức truy hồi đi đến bài toán lớn nhất.

Do mỗi bài toán quy hoạch động đều có đặc thù riêng nên để hiểu rõ cách xác định bài toán và xây dựng bài toán cơ sở và công thức truy hồi, các bạn sẽ được xem ví dụ cụ thể cho một số bài toán quy hoạch động ở phần III phía dưới.

## III. Một số bài toán, ví dụ về quy hoạch động

### 1. Bài toán về dãy con đơn điệu dài nhất

**Bài toán :** Cho dãy số nguyên (a1 , a2, … ,an ). Biết rằng, một dãy con của là một cách chọn ra trong một số phần tử giữ nguyên thứ tự. Hay tìm dãy con gồm các phần tử a[i1], a[i2], … a[ik] với i1<i2<…<ik và a[i1]<a[i2]<…<a[ik].

Hãy tìm ra dãy con tăng có độ dài lớn nhất.

Input :

- Dòng đầu tiên nhập vào số nguyên n.

- Dòng tiếp theo nhập vào n số nguyên từ a[1] đến a[n].

Output:

- In ra độ dài của dãy con dài nhất .

```
Input:
6
1 2 5 4 6 2

Output:
4
```

**Giải thích test ví dụ:** Dãy con dài nhất là dãy A[1] = 1 < A[2] = 2 < A[4] = 4 < A[5] = 6, độ dài dãy này là 4.

**Lời giải:**

Bổ sung thêm 2 phần tử a[0] = - ∞ và a[n+1] =+ ∞. Khi đó dãy con tăng dài nhất chắc chắn sẽ bắt đầu từ a[0] và kết thúc ở a[n+1].

***Xác định bài toán.***

Với mọi i : $0 \le i \le n+1$. Ta sẽ tính L[i] là độ dài của dãy con tăng dài nhất bắt đầu từ vị trí i.

***Bài toán cơ sở.***

L[n+1] sẽ là trạng thái bắt đầu cho thuật toán QHĐ mà được áp dụng cho bài này. Do L[n+1] là độ dài của dãy con tăng dài nhất bắt đầu từ vị trí n+1 => L[n+1]=1.

Từ L[n+1] ta sẽ tiến hành tìm các giá trị của L từ n đến 0.

***Xây dựng công thức truy hồi, tiến hành tính L[n] đến L[0]***

Ta sẽ phải thực hiện tính lần lượt các L[i] với chạy từ n về 0, ta sẽ tính L[i] dựa trên điều kiện L[i+1 ... n+1] đã biết :

Dãy con tăng dài nhất bắt đầu từ a[i] sẽ được thành lập bằng cách lấy a[i] ghép vào đầu một trong số những dãy con tăng dài nhất bắt đầu từ vị trí a[j] đứng sau a[i].

Ta sẽ chọn dãy bắt đầu tại a[j] nào đó mà lớn hơn a[i] ( để đảm bảo tính tăng) và chọn dãy dài nhất để ghép a[i] vào đầu ( để đảm bảo tính dài nhất ) .

Công thức truy hồi : L[i] = max(L[j])  (với j = i+1..n+1 và a[j] >a[i])

Ta sẽ xét tất cả các chỉ số j trong khoảng i+1 đến n+1 mà a[j] > a[i], chọn ra chỉ số jMaxcó L[jMax] lớn nhất.

Lúc đó L[i] sẽ được cập nhật theo L[jMax] : L[i] = L[jMax] +1;

***Code C++:***

```
// Xay dung cong thuc .
    a[0]=-Max;  a[n+1]=Max; // Them 2 phan tu a[0] a[n+1] va day
    L[n+1]=1; // Trang thai ban dau
    for(int i = n; i >= 0 ; i++){
        jMAX=n+1;
        for(int j = i+1 ; j <= n+1 ; j++){
            if(a[j] > a[i] && L[j] > L[jMAX]){
                jMAX = j;
            }
            L[i] = L[jMAX]+1; //Luu do dai day con lon nhat tai a[i]
        }
    }
```

***Thực hiện truy vết***

Ở tất cả các bài toán quy hoạch động, ta luôn có thể truy vết đưa ra trạng thái khi tính được bài toán tối ưu. Và cách lần vết là ta sẽ lần từ con cuối cùng được tính, ở đây là vị trí L[0] = max(L[]) mỗi phần tử luôn được cập nhật theo một vị trí tính trước đó, dựa vào tính chất này mà ta có thể dễ dàng lần vết ra được kết quả. Ở đây ta sẽ lần vết để in ra dãy con tăng dài nhất.

***Code C++ :***

```
void trace(){
    int u = 0 ;  // phan tu cuoi cung hien co
    for (int v = 1;v<=n; v++){
    // duyet cac phan tu lan luot tu 1-> n
    // khong phai tu 1->n+1 do phan tu n+1 luon duoc cap nhat
    // va khong can thiet phai in ra phan tu thu n+1
        if (F[v] == F[u] + 1 && a[u]<a[v]){
            // neu v duoc tinh theo u => ket nap v vao day dang lan vet
            u = v; // phan tu cuoi cung hien co la v => u = v
            cout<<u<<' ';
        }
    }
}
```

***Ví dụ :*** Với a =(5, 2, 3, 4, 9, 10, 5, 6, 7, 8). Hai dãy L được tính theo chiều "Calculating" trước và dựa vào L[] cộng với giá trị mảng a[] ta thực hiện tracing để lấy được dãy a[2] , a[3], a[4], a[7], a[8], a[9], a[10] ⇔ 2, 3, 4, 5, 6, 7, 8 là dãy con tăng lớn nhất

Calculating

| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $a_i$ | $-\infty$ | 5 | 2 | 3 | 4 | 9 | 10 | 5 | 6 | 7 | 8 | $+\infty$ |
| L[i] | 9 | 5 | 8 | 7 | 6 | 3 | 2 | 5 | 4 | 3 | 2 | 1 |
| u | 0 | 0 | (2) | (3) | (4) | 4 | 4 | (7) | (8) | (9) | (10) | x |

*if (L[i] == L[u] +1 && a[i] > a[u]) -> u = i, push(u,kq[])*

Tracing

***Một cách xác định bài toán khác :***

Ở trên ta có thực hiện bổ sung thêm 2 phần tử a[0] = - ∞ và a[n+1] =+ ∞. Khi đó dãy con tăng dài nhất chắc chắn sẽ bắt đầu từ a[0] và kết thúc ở a[n+1].

Bây giờ nếu **đặt bài toán** là : Với mọi i : $0 \le i \le n+1$. Ta sẽ tính L[i] là độ dài của dãy con tăng dài kết thúc tại vị trí i.

Lúc này **bài toán cơ sở** sẽ là L[0] do dãy kết thúc tại vị trí 0 luôn có độ dài bằng 1 chính là phần tử a[0] và kết quả cần tìm chính là L[n+1] : dãy dài nhất khi xét đến phần tử cuối dãy.

Và **công thức truy hồi** lúc này cũng sẽ thay đổi theo bài toán đặt ra :

L[i] = max (L [j]) ( với j<i hay j = 0…i-1 và a[j]<a[i] do tính chất tăng từ vị trí bé đến vị trí lớn).

Các bạn có thể code lại bài này với cách xác định bài toán như trên.

***Nhận xét :***

Bài toán trên cho ta thấy mỗi bài quy hoạch động nếu đặt ra bài toán khác nhau thì cách giải quyết bài toán sẽ khác nhau, và nếu như đặt bài toán sai hoặc chưa đủ sẽ dẫn đến công thức truy hồi sai hoặc không xác định được.

Bài toán trên sử dụng quy hoạch động, ở đây số bài toán con là n và chi phí chuyển mỗi lần tính công thức truy hồi là duyệt tất cả các vị trí của lần duyệt trước nên độ phức tạp là O(n^2).

Các bạn có thể tìm hiểu thêm thuật toán xử lý phần chuyển bài toán tìm L[jMax] với độ phức tạp log(n) bằng cách kết hợp với stack và chặt nhị phân, lúc đó bài toán được giải quyết trong O(n log n)

## 2. Bài toán xâu con chung dài nhất.

**Bài toán :** Cho 2 xâu X, Y với độ dài lần lượt là m và n.

Hãy tìm một xâu S thỏa mãn:

-S là xâu con chung của X, Y( từ X,Y có thể xóa một vài kí tự để biến nó thành xâu S, Ví dụ “abc” là xâu con của “xaxxbyyyczz” vì có thể xóa đi các ký tự của “xaxxbyyyczz” để được “abc”).

-Độ dài của S là lớn nhất.

Ví dụ : X = “abc1def2ghi3” , Y= “abcdefghi123” => Xâu con chung dài nhất là “abcdefghi3” có độ dài là 10

**Lời giải:**

***Xác định bài toán :***

Giả sử xâu X và Y được xét từ vị trí 1-> X.length() và 1-> Y.length() .
Gọi L[i][j] là độ dài xâu con chung dài nhất của 2 xâu : xâu kết thúc tại vị trí i của xâu X và xâu kết thúc tại vị trí j của xâu Y : Hay X[1..i] và Y[1..j].
Kết quả bài toán sẽ là L[n][m] – xâu con chung kết thúc ở n và m ở lần lượt X và Y ( 2 vị trí cuối cùng của 2 xâu ).

***Bài toán cơ sở :***

Ta chỉ quan tâm đến bài toán cơ sở ở những lần tính kết quả đầu tiên vì những vị trí phía sau sẽ được cập nhật lại theo vị trí trước đó.

Bài toán cơ sở : ở phần trên chúng ta đã xét xâu từ vị trí 1 mục đích để khởi tạo biên tức là phần đã được tính trước là L[0][j] và L[i][0] và các giá trị này đều bằng 0 . Vì nếu không xét từ vị trí 1 : ta phải tính được L[0][j] và L[i][0] với j = 1..m và i = 1..n để khởi tạo bài toán cơ sở, đây là một kỹ thuật tạo biên để tạo bài toán cơ sở dễ dàng hơn.

Vậy với X[1..n] và Y[1..m] ta có bài toán cơ sở : L[0][j] = L[i][0] = 0; với mọi (i=0..n và j = 0..m)

***Xây dựng công thức truy hồi.***

Xét L[i][j] – độ dài xâu con chung của X[1..i] và Y[1..j] ta tìm được công thức truy hồi như sau :

Nếu X[i] == Y[j] thì : L[i][j] = L[i-1][j-1]+1 :

Bằng xâu con chung dài nhất phía trước của X[1..i-1] và Y[1..j-1] cộng thêm ký tự vừa mới xét bằng nhau.

Ngược lại – X[i] <> Y[i] thì : L[i][j] = max (L[i-1][j] , L[i][j-1] )

Ở đây 2 ký tự đang xét khác nhau nên thao tác xét i và j sẽ tương ứng với việc thêm ký tự X[i] vào cặp xâu (X[1..i-1] và Y[1..j]) hoặc Y[j] vào (X[1..i] và Y[1..j-1]) nên ta sẽ chọn phương án lớn hơn trong 2 phương án.

***Code C++ :***

```
int solve(){
    for (int i=0; i<=n; i++) L[i][0] = 0;
    for (int j=0; j<=m; j++) L[0][j] = 0; // Khoi tao bai toan co so

    for (int i=1; i<=n;i++){
        for (int j=1;j<=m;j++){
            // Do ta tinh X = X[1..n] ,Y = Y[1..n]
            // nen khi xet X[i] va Y[j] tuong ung voi X[i-1] va Y[i-1]
            if (X[i-1] == Y[j-1]) L[i][j] = L[i-1][j-1] +1;
            else{
                L[i][j] = max (L[i-1][j], L[i][j-1]);
            }
        }
    }
}
```

Bài toán thực hiện với độ phức tạp O(n*m).

***Thực hiện truy vết:***

Tương tự với bài dãy con tăng dài nhất, ta sẽ thực hiện truy vết từ phần tử được tính cuối cùng : L[n][m] , với mỗi phần tử L[i][j] nếu X[i] == Y[j] thì đương nhiên L[i][j] tính theo L[i-1][j-1] và trong trường hợp ngược lại thì nếu L[i][j] == L[i-1][j] thì (i, j) được tính theo (i-1,j). và nếu bằng L[i][j-1] thì (i ,j) đươc tính theo (i,j-1).

```
void trace(){
    int i = n;
    int j = m;
    /// truy vet tu vi tri cuoi cung
    while (i>0 && j>0){
        if (X[i-1] == Y[j-1]){ // ket nap them 1 ky tu
            kq[i-1] = 1;// danh dau vi tri thu i trong X thuoc xau ket qua (1)
            i--;
            j--;
        }
        else{   // khong ket nap them 1 ky tu
            if (L[i][j] == L[i-1][j]) i--;// (i,j) tinh theo (i-1,j)
            else                      j--;// (i,j) tinh theo (i,j-1)
        }
    }
    for (int i=0; i<n; i++){
        if (kq[i]) cout<<X[i]; // (1)
    }
}
```

***Ví dụ :*** Với X = “ABAC” , Y = “ACBC” , bài toán cơ sở chính là hàng 0 và cột 0 đều bằng 0, mỗi phần tử L[i][j] xác định dựa vào 3 giá trị L[i-1][j-1], L[i][j-1] và L[i-1][j] ta có thể thấy dễ hơn là 3 phần tử góc trên đã được tính trong ma trận tính toán. Phần truy vết ta sẽ đi từ F[4][4] về, với mỗi F[i][j] ta biết được ô này được tính theo ô nào phía trên và thực hiện đi lên, mỗi vị trí nếu X[i] bằng Y[j] thì sẽ lấy thêm được một ký tự vào xâu kết quả.

| | | | A | C | B | C | |
|---|---|---|---|---|---|---|---|
| | L[i][j] | 0 | 1 | 2 | 3 | 4 | |
| | 0 | 0 | 0 | 0 | 0 | 0 | |
| A | 1 | 0 | 1 | 1 | 1 | 1 | => "A" |
| B | 2 | 0 | 1 | 1 | 2 | 2 | => "B" |
| A | 3 | 0 | 1 | 1 | 2 | 2 | => Φ |
| C | 4 | 0 | 1 | 2 | 2 | 3 | => "C" |
| | | | | | | Res = | "ABC" |

## 3. Bài toán cái túi – quy hoạch động.

**Bài toán :** Một siêu thị có n gói hàng, gói thứ i có trọng lượng Wi, giá trị Vi. Tên trộm đột nhập vào siêu thị nhưng hắn chỉ có thể mang được trọng lượng M. Tìm các gói hàng tên trộm lấy được để tổng giá trị là lớn nhất.

Bài toán cái túi trước đây chúng ta đã giải quyết theo phương án sinh nhị phân (duyệt toàn bộ) hay đệ quy quay lui+ nhánh cận để giải quyết, tuy nhiên nếu như với những bài có điều kiện tổng khối lượng đủ bé để lưu trữ ta có thể dùng phương pháp quy hoạch động để tìm ra bài toán tối ưu.

**Lời giải:**

***Xác định bài toán :***

Gọi B[i][j] là giá trị lớn nhất khi chọn một số trong i gói hàng đầu tiên (xét đến gói hàng thứ i) với trọng lượng giới hạn bằng j. Khi đó B[n][M]) chính là đáp án cần tìm.

***Bài toán cơ sở :***

B[0][j]=0 (do không chọn gói đồ nào).

***Xây dựng công thức truy hồi.***

Xét gói hàng thứ i :

Tương tự với phương pháp sinh : sẽ có 2 lựa chọn 0 hoặc 1-chọn hoặc không chọn gói hàng i.

Nếu không chọn gói hàng i : F[i][j] = F[i-1][j] ;

Nếu chọn gói hàng i : F[i][j] = F[i-1][j-w[i]] + v[i];

( để chọn thêm gói i ta phải xét i-1 gói phía trước nó và tổng giới hạn là j- w[i] thì khi cộng thêm gói thứ i mới thỏa mãn, đồng thời giá trị F[i][j] sẽ được cộng thêm một lượng v[i] so với F[j-1][j-w[i]]).

Công thức truy hồi F[i][j] = max (F[i-1][j] , F[i-1][j-w[i]] + v[i]);

Đáp án F[n][M]

Độ phức tạp thuật toán : số lượng bài toán n*M , chi phí chuyển O(1) => Độ phức tạp O(n*M).

***Bảng phương án :***

Bảng phương án F gồm n+1 dòng và M +1 cột, trước tiên điền bài toán cơ sở quy hoạch động : Dòng 0 gồm toàn số 0. Sử dụng công thức truy hồi, dùng dòng 0 tính dòng 1, dùng dòng 1 tính dòng 2,… mỗi dòng tính từ 0-> M đến khi hết dòng n ta được F[n][M]

| F | 0 | 1 | 2 | ..... | M |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | ...0... | 0 |
| 1 | | | | | |
| 2 | | | | | |
| ... | ... | ... | ... | ... | |
| n | | | | | |

***Code C++ :***

```
1       for(int i=0;i<=M;i++) b[0][i]=0;
2       for(int i=1;i<=n;i++){
3           for(int j=1;j<=M;j++){
4               if(j-a[i].W>=0) b[i][j]=max(b[i-1][j],a[i].V+b[i-1][j-a[i].W]);
5               else b[i][j]=b[i-1][j];
6           }
7       }
```

***Thực hiện truy vết:***

Tương tự với các bài toán trên, bài toán cái túi thực hiện truy vết dựa vào tính chất ta xác định xem phần tử (i ,j) được cập nhật theo phần tử (i-1,j') nào trước đó. Thực hiện truy vết từ (n,M) đến khi hết dãy.

Code dưới đây dùng mảng kq[] đánh dấu : kq[i] == 1 nếu i được chọn.

```
1  void Trace(){
2      int i=n,j=M;
3      while(i!=0){
4          if(b[i][j]!=b[i-1][j]) {
5              kq[i]=1;
6              j=j-a[i].W;
7          }
8          i=i-1;
9      }
10 }
```

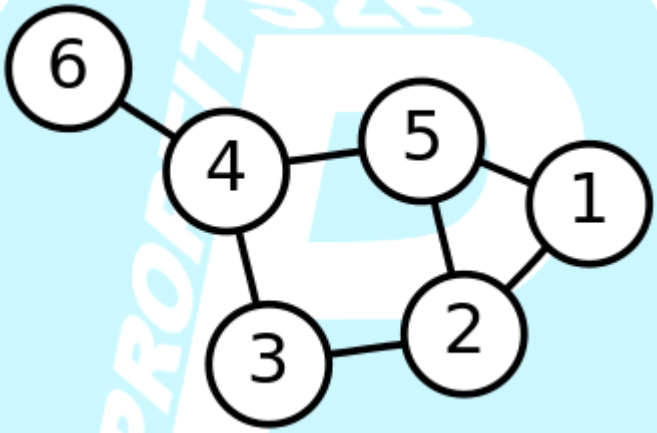
# Bài 6: Đồ thị - Các phương pháp duyệt đồ thị

## I. Lý thuyết cơ bản về đồ thị

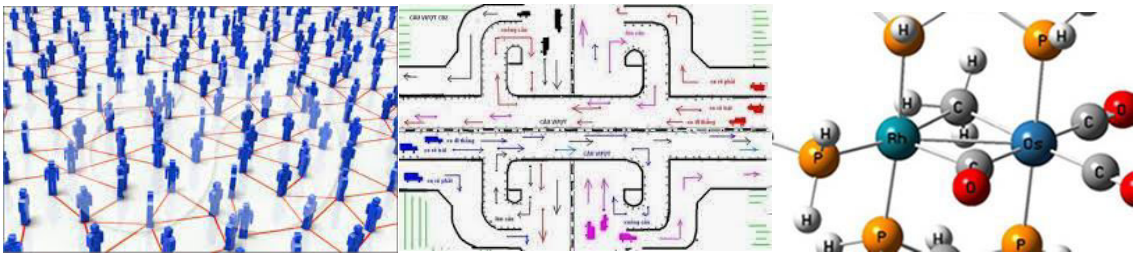
### 1.Định nghĩa đồ thị:

Đồ thị (GRAPH) là một cấu trúc rời rạc gồm các đỉnh và các cạnh nối các đỉnh đó. Được mô tả : G = (V,E)
Trong đó V là tập hợp các đỉnh , E là tập hợp các cạnh – các đường nối giữa các cặp đỉnh.

***Ví dụ :*** trong đồ thị dưới đây :
- Tập đỉnh bao gồm 6 đỉnh V ={1,2,3,4,5,6}
- Tập cạnh gồm 7 cạnh, là các đường nối giữa các cặp đỉnh trong đồ thị
  E ={ (1,2) , (1,5) , (5,2) , (2,3) , (3,4) (4,5) (4,6)}



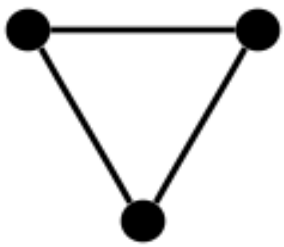

Đồ thị ngày nay được ứng dụng rất rộng trong đời sống. Một số ứng dụng dễ thấy như :
- Mạng xã hội Facebook : mỗi người dung (user) có thể được coi là một đỉnh trong đồ thị, mối quan hệ giữa những user có thể được coi là tập cạnh, dựa vào mối quan hệ đo, Facebook có thể sử dụng những thuật toán đồ thị để quản lý.
- Hay những ví dụ khác trực quan hơn như giao thông, các điểm nút giao thông là các đỉnh, các con đường nối các điểm nút sẽ tạo thành các cạnh,…v…v.. hay các phân tử , nguyên tử liên kết với nhau trong hóa học, ,…v..v..



## 2. Phân loại

Đồ thị được chia làm nhiều loại. Nhưng ứnkg dụng chủ yếu trong tin học gồm 2 loại là **đồ thị có hướng** và **đồ thị vô hướng**.

### a. Đồ thị vô hướng



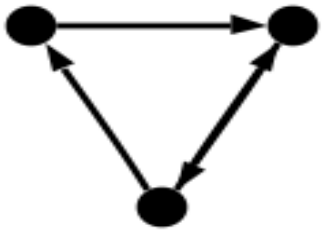
ĐỒ THỊ VÔ HƯỚNG

**Đồ thị vô hướng** hoặc **đồ thị** *G* là một cặp không có thứ tự *(unordered pair) G*:=(*V*, *E*), trong đó

- *V*, tập các **đỉnh** hoặc **nút**,
- *E*, tập các cặp không thứ tự chứa các đỉnh phân biệt, được gọi là **cạnh**.Hai đỉnh thuộc một cạnh được gọi là các **đỉnh** đầu cuối của cạnh đó.

**Hiểu một cách đơn giản** : Chẳng hạn ta có 1 đồ thị vô hướng biểu diễn đường đi giữa các thành phố. Nếu có đường đi từ thành phố A tới thành phố B thì **sẽ có** đường đi từ thành phố B trở về thành phố A.

### b. Đồ thị có hướng



ĐỒ THỊ CÓ HƯỚNG

**Đồ thị có hướng** *G* là một cặp có thứ tự *G*:=(*V*, *A*), trong đó

- *V*, tập các **đỉnh** hoặc **nút**,
- *A*, tập các cặp có thứ tự chứa các đỉnh, được gọi là các **cạnh có hướng** hoặc **cung**. Một cạnh *e* = (*x*, *y*) được coi là có hướng **từ** *x* **tới** *y*; *x* được gọi là **điểm đầu/gốc** và *y* được gọi là **điểm cuối/ngọn** của cạnh.

**Hiểu một cách đơn giản** : Chẳng hạn ta có 1 đồ thị có hướng biểu diễn đường đi giữa các thành phố. Nếu có đường đi từ thành phố A tới thành phố B thì **chưa chắc sẽ có** đường đi từ thành phố B trở về thành phố A. Còn tùy thuộc vào dữ liệu bài toán cho dựa vào các mũi tên chỉ hướng trong đồ thị.

## 3. Các khái niệm

### a. Cạnh liên thuộc, đỉnh kề, bậc :

- Với mỗi cạnh e =(u,v) ϵ E thì ra nói đỉnh u và đỉnh v kề nhau , và e là cạnh liên thuộc giữa 2 đỉnh u và v
- Bậc của đỉnh v ϵ V ký hiệu deg(v) được tính bằng số cạnh liên thuộc với đỉnh v trong đồ thị. Trong đơn đồ thị, số đỉnh kề với v cũng chính bằng số cạnh liên thuộc với v và bằng với bậc của v.
- Đối với đồ thị có hướng, mỗi cạnh e =(u,v) : ta có khái niệm đỉnh u đi đến v hay đỉnh v đi ra từ u. Và đối với bậc trong đồ thị có hướng, ta sẽ có 2 khái niệm là bán bậc ra và bán bậc vào : deg+(v) và deg-(v) : deg+(v) là số cung đi ra khỏi đỉnh v và deg-(v) là số cung đi vào đỉnh v.

### b. Đường đi và chu trình :

- Một đường đi có độ dài p bao gồm tập các đỉnh P = {v[0] , v[1] , … , v[p]} trong đó ( v[i-1] , v[i] ) với i = 1..p là một cạnh thuộc G (V,E). Có thể hiểu là cách di chuyển qua các đỉnh bằng cách đi qua các cạnh của đồ thị.
- Một chu trình là một đường đi nếu v[0] bằng v[p] hay có thể hiểu là đường đi bắt đầu từ một đỉnh đi qua một số đỉnh khác và quay về đỉnh bắt đầu. Một chu trình đơn giản khi v1,v2,… vp khác nhau đôi một.

### c. Liên thông :

- Một đồ thị vô hướng liên thông khi với mọi cặp đỉnh (u,v) bất kỳ luôn tồn tại một đường đi từ u đến v
- Một đồ thị có hướng, được gọi là liên thông mạnh khi mọi cặp đỉnh (u,v) luôn tồn tại đường đi từ u đến v và đường đi từ v đến u.
- Một đồ thị có hướng, được gọi là liên thông yếu khi với mỗi cặp đỉnh (u,v) ta có thể đi từ u đến v hoặc từ v về u.

## 4. Các cách biểu diễn và lưu trữ đồ thị

Hiển nhiên, khi muốn biểu diễn một đồ thị, ta phải biểu diễn được đủ các nút và các mối liên hệ giữa các nút đó.

Ta có thể dễ dàng lưu trữ và biểu diễn các nút thông qua một mảng (có thể là mảng tĩnh hoặc mảng động), nhưng với các cạnh thì không đơn giản như thế.

Về nguyên tắc, có 2 giải pháp để biểu diễn các cạnh:

1. Liệt kê tất cả các cạnh của từng nút.
2. Liệt kê xem giữa 2 nút bất kỳ có đường nối với nhau hay không.
3. Danh sách cạnh

Ứng với 2 giải pháp này, ta cũng có 2 cách biểu diễn đồ thị khi lập trình, đó là sử dụng ma trận liền kề (adjacency matrix) hoặc danh sách liền kề (adjacency list), hoặc một cách khác ít dung hơn ( dùng cho bài toán cây khung) là sử dụng danh sách cạnh.

### a. Lưu trữ đồ thị bằng "Ma trận liền kề" (adjacency matrix)

**Định nghĩa**

Phương pháp sử dụng ma trận liền kề là phương pháp liệt kê các nút và mối liên hệ giữa chúng thông qua một ma trận A kích thước N x N (với N là số nút của đồ thị). Đối với đồ thị không trọng số : trong ma trận này, phần tử A(I, J) sẽ có giá trị là 0 nếu không có đường nối trực tiếp từ nút I tới nút J trong đồ thị. Ngược lại, giá trị của phần tử này sẽ là 1.

**Xây dựng**

```cpp
#include <iostream>
using namespace std;
 // Undirected graph
int main() {
    int node, edge_param1, edge_param2;
    cin >> node;
    int matrix[node][node];
    for (int i=0; i<node; i++) {
        for (int j=0; j<node; j++) {
            matrix[i][j] = 0;
        }
    } // Initializing matrix values.
    while (cin >> edge_param1 >> edge_param2) {
        matrix[edge_param1][edge_param2] = 1;
        matrix[edge_param2][edge_param1] = 1;
    }
    for (int i=0; i<node; i++) {
        for (int j=0; j<node; j++) {
            cout << matrix[i][j] << " ";
        }
        cout << endl;
    }
    return 0;
}
```

stdin

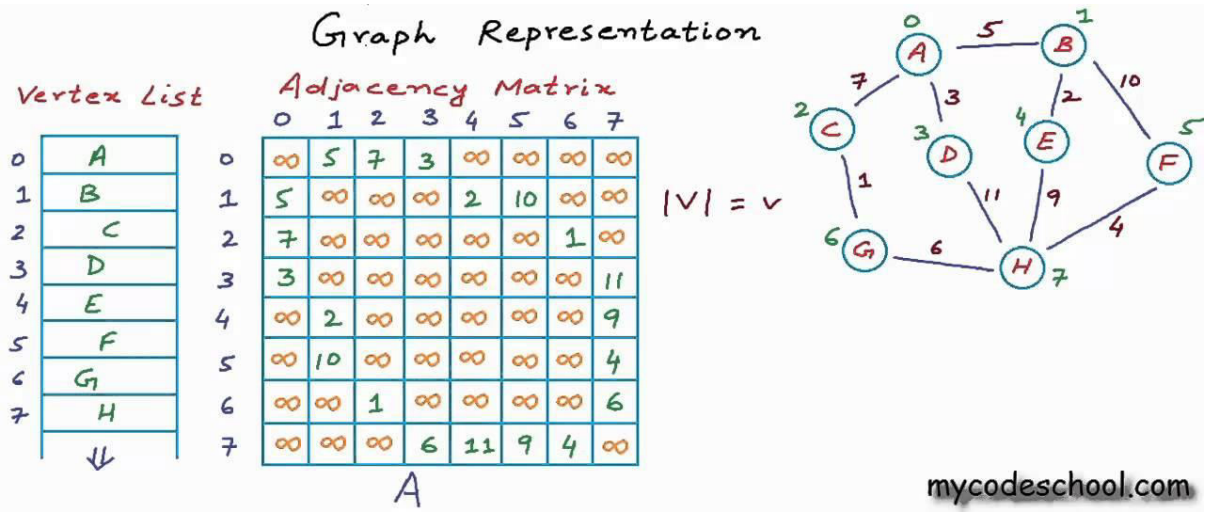
```
8
0 3
2 6
6 7
1 5
0 1
0 2
3 7
5 7
4 7
1 4
```

stdout

```
0 1 1 1 0 0 0 0
1 0 0 0 1 1 0 0
1 0 0 0 0 0 1 0
1 0 0 0 0 0 0 1
0 1 0 0 0 0 0 1
0 1 0 0 0 0 0 1
0 0 1 0 0 0 0 1
0 0 0 1 1 1 1 0
```

Trên hình là một ví dụ cho việc biến đổi một đồ thị v\ô hướng về dạng ma trận kề. Lưu ý là từ sau ví dụ này, mọi đồ thị được xét đến đều không có khuyên, cho nên đường chéo chính của ma trận luôn bằng 0 (trường hợp ví dụ này thì có 1 khuyên ở nút D). Nếu đồ thị là vô hướng, thì hiển nhiên ma trận liền kề đối xứng nhau qua đường chéo chính.

- Đối với đồ thị có trọng số : nếu các cạnh của đồ thị có trọng số thì thay vì nhận giá trị 1 ở các nút có liên kết, các phần tử khác 0 của ma trận liền kề sẽ nhận giá trị đúng bằng trọng số. Những cặp đỉnh không có đường nối ta có thể gán giá trị trọng số rất lớn biểu hiện cho việc không có đường đi.




***Chú ý:***

1. Với đồ thị có hướng, khi lập trình phải nắm rõ quy ước của chính mình đối với dữ liệu nhập vào. Chẳng hạn, nếu input cho cạnh x->y thì chỉ có 1 trong 2 lệnh sau được thực hiện: hoặc là gán matrix[x][y] = 1 hoặc gán matrix[y][x] = 1.
2. Lệnh (while (cin >> edge_param1 >> edge_param2)) có tác dụng đọc dữ liệu cho tới hết luồng stdin/istream. Để an toàn, khi bài toán cho sẵn số lượng cạnh thì nên dùng một lệnh lặp mà điều kiện có thể kiểm soát bằng mắt thường được.

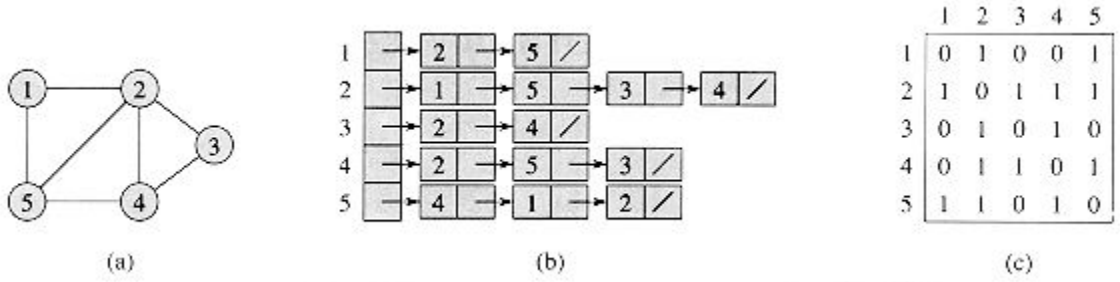
Theo ví dụ trên, ta có thể nhận thấy một số đặc điểm của việc sử dụng ma trận liền kề:

- Độ phức tạp O(1) do việc kiểm tra tính liền kề của 2 phần tử chỉ thông qua kiểm tra giá trị 1 phần tử của ma trận.
- Dung lượng bộ nhớ O(n^2) là RẤT LỚN, do vậy chỉ nên sử dụng ma trận liền kề khi thỏa mãn ít nhất 1 trong 2 điều kiện sau:
    - Số lượng nút của đồ thị nhỏ (khuyến nghị tối đa là 10^4)
    - Đồ thị có độ đặc khít cao (nhằm tránh lãng phí tài nguyên bộ nhớ cho những cặp nút không có đường nối). Bài toán trên cũng là một ví dụ cho đồ thị kém đặc khít.

## a.Lưu trữ, biểu diễn đồ thị bằng danh sách kề

**Định nghĩa**

Phương pháp sử dụng danh sách kề là phương pháp liệt kê các nút và mối liên hệ giữa chúng qua chuỗi các danh sách, mỗi danh sách biểu diễn các nút kế cận nút đang xét trong đồ thị.




Hình trên là so sánh tương quan giữa đồ thị hình học, đồ thị biểu diễn bởi danh sách kề và ma trận liền kề. Có 5 danh sách, danh sách thứ i sẽ lần lượt trỏ đến các đỉnh kề với đỉnh i.

**Xây dựng danh sách kề trong chương trình máy tính**

Có nhiều cách để mô phỏng danh sách kề cho một đồ thị bằng phương pháp lập trình. Ở đây, tài liệu chỉ đề cập tới 3 phương pháp:

### *a.1. Sử dụng danh sách liên kết*

Phương pháp này hầu như không được sử dụng đến, do sự quá tải bộ nhớ / thời gian và những rủi ro có thể gặp phải khi sử dụng bộ nhớ động và con trỏ.

### *a.2. Sử dụng mảng các vector*

Ý tưởng của cách xây dựng này là: khởi tạo một mảng gồm n vector, mỗi vector ứng với 1 nút và chứa số hiệu của các nút liền kề với nút được xét.

Chú ý:

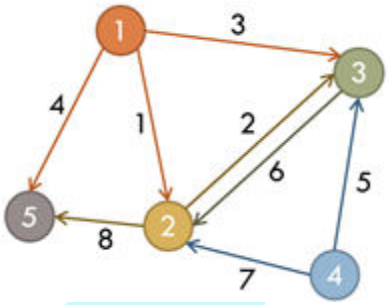
Trong trường hợp đồ thị có hướng thì chỉ có 1 chiều dịch chuyển trên mỗi cạnh (từ u tới v) nên sẽ không có lệnh gán u vào vector phần tử liền kề của v.

Ý tưởng này tương đối dễ hiểu, trực diện, dễ lập trình và có thể đảm bảo không gặp các lỗi phát sinh về bộ nhớ (như các vấn đề về bộ nhớ động, segmentation fault [SIGSEGV], v.v. …), song thời gian xử lý chương trình lại tương đối chậm. (do mỗi truy vấn để tìm kiếm phần tử phải thông qua duyệt toàn bộ 1 vector, với các đồ thị có lượng nút và đường nối lớn thì các truy vấn này có tốc độ rất chậm).

Dưới đây là đoạn chương trình nhập vào danh sách các cạnh của đồ thị và chúng ta thực hiện xây dựng các danh sách kề ke[u] là danh sách các đỉnh kề với đỉnh u.

```
void readf(){
    int n , m;
    cin >> n >> m;`
    vector <int>  ke[n+1];
    for (int i=1; i<=m; i++){
        cin>>u >>v ;
        ke[u].push_back(v) ; // push dinh v vao danh sach dinh ke voi u
        ke[v].push_back(u) ; // do thi vo huong => (u,v) + (v,u)
    }
}
```

Trong hình bên, đồ thị có hướng, ta chỉ push theo hướng của cung chứ không thực hiện push vào 2 chiều. Ở đây :




ke[1] = (2, 3, 5)

ke[2] = (5 , 3)

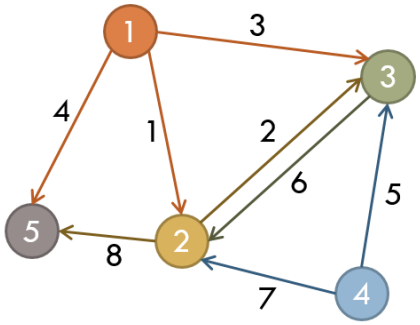
ke[3] = (2)

ke[4] = (2, 3)

Đối với đồ thị có trọng số, danh sách các đỉnh kề thay vì lưu mình chỉ số đỉnh kề ta có thể thêm vào giá trị trọng số của cạnh bằng cách khai báo struct : ke[u].push_back( NODE ( v , c)); trong đó v là đỉnh kề và c là trọng số.

### *a.3. Danh sách liên thuộc sử dụng mảng*

Dưới đây là một cách biểu diễn ma trận nữa, không phải dạng danh sách kề, nhưng nhìn chung nó được lưu trữ và thời gian truy vấn tương đồng với danh sách kề, được gọi là danh sách liên thuộc, sử dụng 2 mảng để truy vấn đến các cạnh trong đồ thị.




| ID | To | Next Edge ID |
|---|---|---|
| 1 | 2 | - |
| 2 | 3 | - |
| 3 | 3 | 1 |
| 4 | 5 | 3 |
| 5 | 3 | - |
| 6 | 2 | - |
| 7 | 2 | 5 |
| 8 | 5 | 2 |

| From | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Last Edge ID | 4 | 8 | 6 | 7 | - |

Hình trên là một ví dụ trực quan cho cách biểu diễn danh sách kề bằng mảng. Sẽ có 2 mảng được khởi tạo: 1 mảng lưu trữ các đoạn nối [bảng trên] theo cấu trúc: số hiệu đoạn nối (ID), số hiệu nút đích đến của đoạn nối (To), cạnh được duyệt ngay trước nó (xét theo chiều rộng)* (Next Edge ID), và 1 mảng lưu trữ các nút [bảng dưới] theo cấu trúc: số hiệu nút (From), cạnh cuối cùng đi từ nút đó được duyệt (xét theo chiều rộng) (Last Edge ID).

Thuật toán này khiến cho việc thay đổi các thành phần bên trong đồ thị trở nên cực kỳ phức tạp, bởi đôi khi sự thay đổi một yếu tố sẽ đi liền với sự thay đổi hàng loạt các giá trị Last Edge, Next Edge. Tuy nhiên, điểm mạnh đáng kể của cách thực hiện này chính là dung lượng thấp Θ(N+E) và thời gian truy vấn tương đối nhanh.

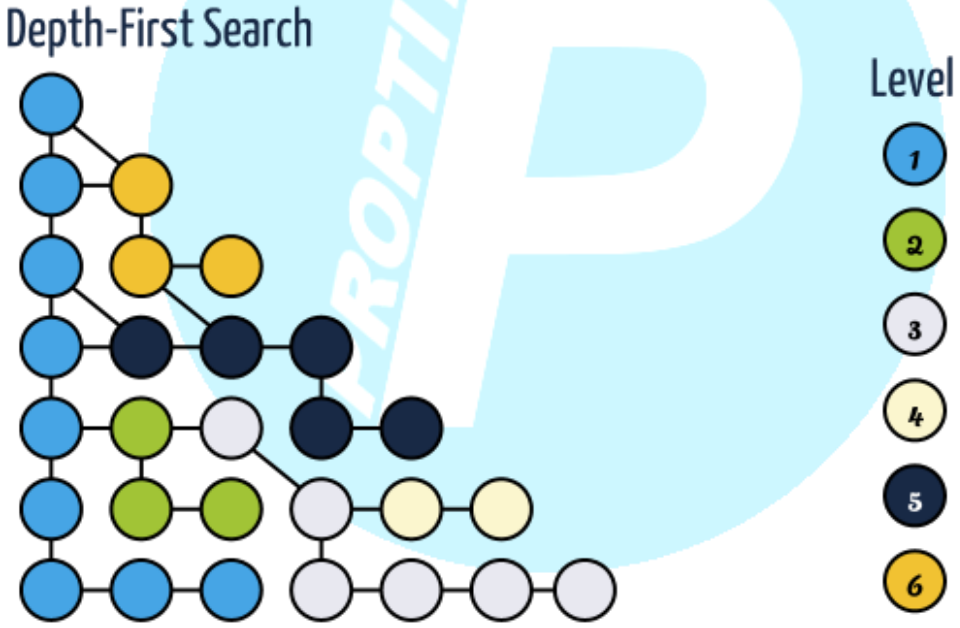
Kết hợp với đệ quy hoặc các cấu trúc dữ liệu (stack / queue) thì ta có thể dễ dàng duyệt qua tất cả các phần tử với tốc độ nhanh trắng. Độ phức tạp O(E).

### b. Danh sách cạnh :

Một cách biểu diễn đồ thị nữa thường được sử dụng trong bài toán tìm cây khung nhỏ nhất bằng thuật toán Kruskal, ta chỉ quan tâm đến các cạnh mà không cần duyệt các cạnh kề của mỗi đỉnh. Ta chỉ cần lưu trữ một mảng các cặp (u, v) đây cũng chính là số cạnh mà đề bài yêu cầu nhập vào nên đương nhiên là cách lưu trữ tối ưu bộ nhớ nhất. Tuy nhiên sử dụng danh sách cạnh tốn rất nhiều thời gian truy vấn, mỗi cặp đỉnh u, v chúng ta đều phải thực hiện duyệt m cạnh trong danh sách. Phương pháp lưu trữ bằng danh sách cạnh chỉ được sử dụng trong một số bài toán đặc thù.

## II.Các thuật toán duyệt đồ thị không trọng số.

Trong các bài toán đồ thị, một trong những yêu cầu căn bản nhất là tra cứu toàn bộ đồ thị mà mỗi nút chỉ đi qua 1 lần. Với yêu cầu này, ta có 2 kiểu thuật toán cơ bản là duyệt theo chiều sâu (Depth-First Search | DFS) và duyệt theo chiều rộng (Breadth-First Search |




BFS)

### 1.Thuật toán duyệt đồ thị theo chiều sâu (Depth-First Search - DFS)

**Định nghĩa**

DFS (Depth-First Search) là tên gọi cho thuật toán tìm kiếm trong đồ thị dựa theo ưu tiên chiều sâu. Có nghĩa là, xuất phát từ một nút, ta bắt đầu duyệt đến tận cùng từng nhánh tỏa ra từ nút đó rồi mới chuyển sang nhánh tiếp theo, rồi nút tiếp theo, v.v. … Dễ hiểu hơn thì cách di chuyển trên đồ thị cũng giống như tư tưởng tham lam của con người : cứ đi

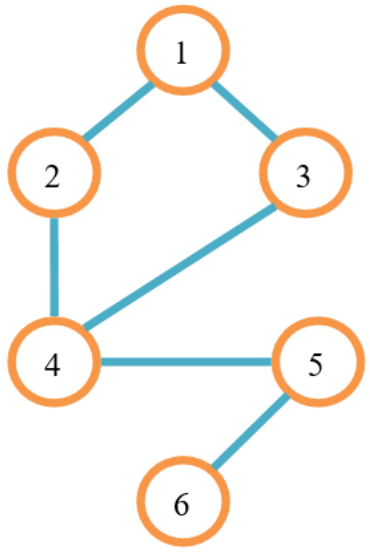
sâu vào đến khi không còn đi được nữa rồi mới quay lại ngã rẽ gần nhất để thử rẽ đường khác.

Trên hình là một ví dụ của DFS, với thứ tự ưu tiên của các nhánh được duyệt theo thứ tự từ 1 tới 6. Việc chọn nhánh duyệt trước và nhánh duyệt sau tùy thuộc vào input và cách phân phối các đường nối trong danh sách / ma trận kề.

## Các cách cài đặt DFS

### *Bài toán cụ thể :*

Cho đồ thị như hình vẽ: Bao gồm các đỉnh từ 1 đến 6 được nối với nhau bởi các đường màu xanh.




Yêu cầu: Duyệt tất cả các đỉnh của đồ thị sao cho mỗi đỉnh viếng thăm đúng 1 lần.

### *Áp dụng DFS (Depth First Search)*

Thuật toán tìm kiếm theo chiều xâu (*DFS*) là bắt đầu từ một đỉnh bất kì ***v*** nào đó + đánh dấu đỉnh ***v*** đã được duyệt, chọn một đỉnh ***u*** bất kỳ kề với ***v*** và lấy nó làm đỉnh duyệt tiếp theo.

Ở đây nếu chọn đỉnh 1 đầu tiên: Ta có cách duyệt :

1 → 2 ; 2→ 4 ; 4→3 ; 4→ 5; 5→ 6; và thứ tự các đỉnh được duyệt lần lượt là :
1 , 2 , 4 , 3 , 5 , 6.

### *Biểu diễn đồ thị :*

Ke(1) = {2, 3};
Ke(2) = {1, 4};
Ke(3) = {1, 4};
Ke(4) = {2, 3, 5};
Ke(5) = {4, 6};
Ke(6) = {5};

Ở đây chúng ta sẽ biểu diễn đồ thị bằng danh sách kề. Do thông thường, input đề bài luôn đọc vào được tất cả các cạnh, nên sử dụng danh sách kề cho bài toán duyệt đồ thị là hợp lý nhất, để tổng số cạnh duyệt qua đúng bằng số cạnh nhập vào.

***Cài đặt thuật toán DFS :*** DFS thông thường có 2 cách cài đặt : sử dụng đệ quy quay lui và cách thứ 2 là sử dụng cấu trúc Stack để khử đệ quy.

## a. Cài đặt thuật toán bằng đệ quy – quay lui :

- ***Nhập dữ liệu biểu diễn đồ thị***

Input yêu cầu nhập vào n,m là số đỉnh và số cạnh, m dòng sau là m cặp cạnh
Lưu trữ đồ thị bằng danh sách kề như trên. Đồng thời khởi tạo mảng đánh dấu các đỉnh ( chưa xét).

```
 6  int const nMax = 1000006;
 7  vector <int> ke[nMax] ;
 8  int n , m ;
 9  bool chuaxet[nMax] ;
10
11  void nhap(){
12      cin>>n >> m; // so dinh va so canh
13      for (int i=1; i<= m ; i++){
14          int u,v;
15          cin>> u >> v;
16          ke[u].push_back(v); // push dinh v vao danh sach ke voi dinh u
17          ke[v].push_back(u); // push dinh u vao danh sach ke voi dinh v
18      }
19
20      for (int i=1; i<=n; i++) chuaxet[i] = true ;
21  }
```

- ***Cài đặt hàm đệ quy DFS :***

Hàm DFS_Dequy(u) sẽ thực hiện duyệt đồ thị từ đỉnh u đi sâu vào đến những đỉnh chưa được xét kề với u, mỗi lần đi qua đỉnh nào thì đánh dấu lại vào danh sách đã xét .Thực hiện đi sâu vào tính từ đỉnh mới nhất cho đến khi không còn đỉnh nào để đi sâu vào nữa, chương trình sẽ quay lui lại các đỉnh (ngã rẽ) trước đó để thực hiện đi sâu theo lối rẽ khác.

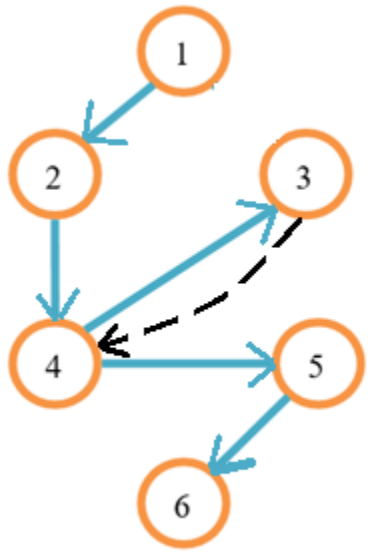
***Code C++ :***

```
19  void DFS_Dequy(int u ){
20      int v;
21      cout << u ;
22      chuaxet[u] = false; // danh dau da xet
23      for (int i = 0 ; i<ke[u].size() ; i++ ){ // Duyet danh sach ke
24          v = ke[u][i] ;
25          if (chuaxet(v)){
26              DFS_Dequy(v); // di sau vao
27          }
28      }
29  }
```

- ***Kiểm nghiệm thuật toán***

| Các đỉnh có thể đi đến | Đỉnh duyệt | Đã duyệt chuaxet[u]=FALSE | Chưa duyệt chuaxet[u]=TRUE |
|---|---|---|---|
| | *DFS_Dequy ( )* | o | 1, 2, 3, 4, 5, 6 |
| 1, 2, 3, 4, 5, 6 | *DFS_Dequy (1)* | 1 | 2, 3, 4, 5, 6 |
| 2, 3 | *DFS_Dequy (2)* | 1, 2 | 3, 4, 5, 6 |
| 4 | *DFS_Dequy (4)* | 1, 2, 4 | 3, 5, 6 |
| 3, 5 | *DFS_Dequy (3)* | 1, 2, 4, 3 | 5, 6 |
| | *DFS_Dequy (5)* | 1, 2, 4, 3, 5 | 6 |
| 6 | *DFS_Dequy (6)* | 1, 2, 4, 3, 5, 6 | o |

**Giải thích:**




- Ở đây, bắt đầu từ đỉnh 1 : 1 -> 2 -> 4 -> 3 . Đến 3 hết đỉnh kề và thực hiện quay lui về 4 tiếp tục duyệt tiếp: 4 -> 5 - > 6, đến 6 có thể dừng thuật toán do đã thăm đủ n đỉnh hoặc thực hiện quay lui về, chương trình sẽ tự động quay lui và không thực hiện đi sâu nữa do đã đi qua hết các đỉnh. Dưới đây là giải thích cách hoạt động :
- Giả sử chọn đỉnh 1: Duyệt (1) và *DFS_Dequy (1)*.
- For (1-> 6) tìm đỉnh chưa được đánh dấu và có thể đi đến từ 1. Gặp 2 thỏa mãn: Duyệt (2) và *DFS_Dequy (2)*.
- For (1-> 6) tìm đỉnh chưa được đánh dấu và có thể đi đến từ 2. Gặp 4 thỏa mãn: Duyệt (4) và *DFS_Dequy (4)*.
- For (1-> 6) tìm đỉnh chưa được đánh dấu và có thể đi đến từ 4. Gặp 3 thỏa mãn: Duyệt (3) và *DFS_Dequy (3)*.
- For (1-> 6) tìm đỉnh chưa được đánh dấu và có thể đi đến từ 3. Không tìm thấy -> Dừng *DFS_Dequy (3)*.
- Tiếp tục *DFS_Dequy (4)*. Gặp 5 thỏa mãn: Duyệt (5) và *DFS_Dequy (5)*.
- For (1-> 6) tìm đỉnh chưa được đánh dấu và có thể đi đến từ 5. Gặp 6 thỏa mãn: Duyệt (6) và *DFS_Dequy (6)*.
- For (1-> 6) tìm đỉnh chưa được đánh dấu và có thể đi đến từ 6. Không tìm thấy -> Dừng *DFS_Dequy (6)*.
- Tiếp tục *DFS_Dequy (5)*. Không tìm thấy -> Dừng *DFS_Dequy (5)*.
- Tiếp tục *DFS_Dequy (4)*. Không tìm thấy -> Dừng *DFS_Dequy (4)*.
- Tiếp tục *DFS_Dequy (2)*. Không tìm thấy -> Dừng *DFS_Dequy (2)*.

- Tiếp tục *DFS_Dequy (1).* Không tìm thấy -> Dừng *DFS_Dequy (1).*
    - ***Hoàn thiện Code :***

```cpp
#include <vector>
#include <iostream>

using namespace std;

int const nMax = 1000006;
vector <int> ke[nMax] ;
int n , m ;
bool chuaxet[nMax] ;

void nhap(){
    cin>>n >> m; // so dinh va so canh
    for (int i=1; i<= m ; i++){
        int u,v;
        cin>> u >> v;
        ke[u].push_back(v); // push dinh v vao danh sach ke voi dinh u
        ke[v].push_back(u); // push dinh u vao danh sach ke voi dinh v
    }

    for (int i=1; i<=n; i++) chuaxet[i] = true ;
}

void DFS_Dequy(int u ){
    int v;
    cout << u ;
    chuaxet[u] = false; // danh dau da xet
    for (int i = 0 ; i<ke[u].size() ; i++ ){ // Duyet danh sach ke`
        v = ke[u][i] ;
        if (chuaxet[v]){
            DFS_Dequy(v); // di sau vao
        }
    }
}

int main(){
    nhap();
    DFS_Dequy(1); // duyet do thi bat dau tu dinh 1
}
```

### b. Cài đặt thuật toán bằng cấu trúc dữ liệu Stack :

Như chúng ta đã biết một ứng dụng của Stack là khử đệ quy, đối với bài toán duyệt đồ thị bằng thuật toán DFS, khi số đỉnh của đồ thị rất lớn ( cỡ 10^6) thì sẽ có một số vấn đề về tràn bộ nhớ đệ quy và để cải thiện điều này thì sử dụng Stack là một cách rất tốt.

- ***Nhập dữ liệu đồ thị*** *: ( Giống như phần cài đặt DFS _ Đệ quy )*

- ***Cài đặt DFS_Stack :***

```
void DFS_Stack(int u ){
    stack <int> s; // khai bao stack
    s.push(u) ; // push dinh dau tien vao stack
    chuaxet[u] = false ; // danh dau da xet dinh u
    cout<< u <<' ';
    while (!s.empty()){ // trong khi stack con chua dinh dem di duyet tiep
        u = s.top(); s.pop(); // lay dinh o dinh stack ra
        for (int i= 0 ; i<ke[u].size(); i++){
            int v = ke[u][i];  // duyet cac dinh trong danh sach ke cua dinh u
            if (chuaxet[v]){
                cout<< v<< ' ';
                chuaxet[v] = false ;
                // Neu di den duoc v thi push (v) vao stack
                // tuy nhien giong voi cach hoat dong cua pp dequy, phai co buoc
                // quay lui lai dinh u, nen ta thuc hien push lai dinh u vao
                s.push(u);
                s.push(v);
            }
        }
    }
}
```

Trên tư tưởng khử đệ quy, mỗi khi pop() 1 đỉnh ra, nếu đỉnh đó còn có thể đi sâu vào ta phải thực hiện push lại đỉnh đó vào để lần pop phía sau có thể đem đỉnh đó rẽ nhánh sang nhánh khác – như tư tưởng quay lui.

- ***Kiểm nghiệm thuật toán cho ví dụ :***

| STT | Trạng thái stack | | Các đỉnh được duyệt |
|---|---|---|---|
| 1 | 1 | | 1 |
| 2 | 1, 2 | | 1, 2 |
| 3 | 1, 2, 3 | | 1, 2, 3 |
| 4 | 1, 2, 3, 4 | (pop(4) ) | 1, 2, 3, 4 |
| 5 | 1, 2, 3, 5 | | 1, 2, 3, 4, 5 |
| 6 | 1, 2, 3, 5, 6 | (pop(6) ) | 1, 2, 3, 4, 5, 6 |
| 7 | 1, 2, 3, 5 | (pop(5) ) | 1, 2, 3, 4, 5, 6 |
| 8 | 1, 2, 3 | (pop(3) ) | 1, 2, 3, 4, 5, 6 |
| 9 | 1, 2 | (pop(2) ) | 1, 2, 3, 4, 5, 6 |
| 10 | 1 | (pop(1) ) | 1, 2, 3, 4, 5, 6 |
| 11 | Rỗng ➔ kết thúc | | 1, 2, 3, 4, 5, 6 |

- ***Hoàn thiện Code :***

```cpp
#include <bits/stdc++.h>
using namespace std;
int const nMax = 1000006;
vector <int> ke[nMax] ;
int n , m ;
bool chuaxet[nMax] ;
void nhap(){
    cin>>n >> m; // so dinh va so canh
    for (int i=1; i<= m ; i++){
        int u,v;
        cin>> u >> v;
        ke[u].push_back(v); // push dinh v vao danh sach ke voi dinh u
        ke[v].push_back(u); // push dinh u vao danh sach ke voi dinh v
    }

    for (int i=1; i<=n; i++) chuaxet[i] = true ;
}
void DFS_Stack(int u ){
    stack <int> s; // khai bao stack
    s.push(u) ; // push dinh dau tien vao stack
    chuaxet[u] = false ; // danh dau da xet dinh u
    cout<< u <<' ';
    while (!s.empty()){ // trong khi stack con chua dinh dem di duyet tiep
        u = s.top(); s.pop(); // lay dinh o dinh stack ra
        for (int i= 0 ; i<ke[u].size(); i++){
            int v = ke[u][i];  // duyet cac dinh trong danh sach ke cua dinh u
            if (chuaxet[v]){
                cout<< v<< ' ';
                chuaxet[v] = false ;
                s.push(u);
                s.push(v);
            }
        }
    }
}
int main(){
    nhap();
    DFS_Stack(1); // duyet do thi bat dau tu dinh 1
}
```

## c. Ứng dụng cho DFS – Bài toán tìm thành phần liên thông

**Định nghĩa thành phần liên thông:**

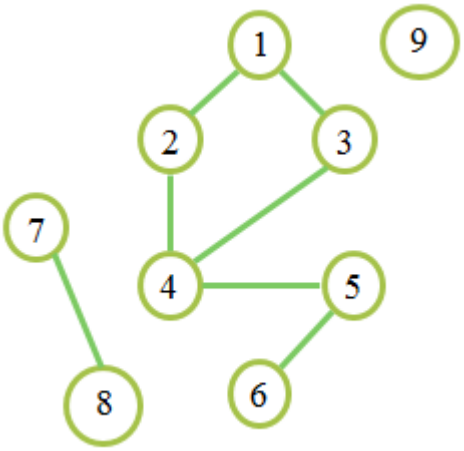
- Đồ thị vô hướng được coi là liên thông nếu luôn tìm được đường đi giữa hai đỉnh bất kỳ của nó.

- Trong trường hợp đồ thị G không liên thông, ta có thể phân ra G thành một số đồ thị con liên thông mà chúng đôi một không có đỉnh chung. Mỗi đồ thị con như vậy là một thành phần liên thông của G. Như vậy, đồ thị liên thông khi và chỉ khi số thành phần liên thông của nó là 1.

- Đối với đồ thị vô hướng, đường đi từ đỉnh ***u*** đến đỉnh ***v*** cũng giống như đỉnh đường đi từ đỉnh ***v*** đến đỉnh ***u***. Chính vì vậy, nếu tồn tại $u \in V$ sao cho ***u*** có đường đi đến tất cả các đỉnh còn lại của đồ thị thì ta kết luận được đồ thị là liên thông.

**Ví dụ :**

Cho đồ thị như hình vẽ xác định số thành phần liên thông:




9

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Dễ thấy ta có 3 thành phần liên thông đó là: {1, 2, 3, 4, 5, 6}; {7, 8} và {9};

**Áp dụng *DFS* tìm thành phần liên thông:**

- ***Ý tưởng:***

Một đồ thị có thể liên thông hoặc không liên thông. Nếu đồ thị liên thông thì số thành phần liên thông của nó là 1. Điều này tương đương với phép duyệt theo thứ tự *DFS (u)* được gọi đến đúng 1 lần.

- ***Mô tả thuật toán:***

Nếu đồ thị không liên thông (số thành phần liên thông lớn hơn 1) chúng ta có thể tách chúng thành những đồ thị con liên thông. Điều này cũng có nghĩa là trong phép duyệt đồ thị, số thành phần liên thông của nó đúng bằng số lần gọi tới thủ tục *DFS ( )*. Để xác định số các thành phần liên thông của đồ thị, chúng ta sử dụng thêm biến solt để ghi nhận các đỉnh của một thành phần thông.

- ***Hoàn thiện Code :***

```
#include <iostream>
using namespace std;

#define Max 50
#define TRUE 1
#define FALSE 0

int A[Max][Max], n, chuaxet[Max];

void khoitao () {
    cin>>n; //Nhap kich thuoc mang A;
    for (int i=1; i<=n; i++) {
        chuaxet[i]=TRUE;    //Cho tat ca cac dinh chua duoc xet;
        for (int j=1; j<=n; j++) {
            cin>>A[i][j];   //Nhap thong tin mang A;
        }
    }
}

void DFS_TPLT (int u) {
    int v;
    cout<<u<<" ";   //In ra cac dinh DFS duyet;
    chuaxet[u]=FALSE;   //Danh dau u da duyet;
    for (int v=1; v<=n; v++) {
        if (A[u][v]==1 && chuaxet[v]==TRUE) {
            DFS_TPLT (v);   //Neu u co the den v va v chua duoc duyet thi di toi v;
        }
    }
}

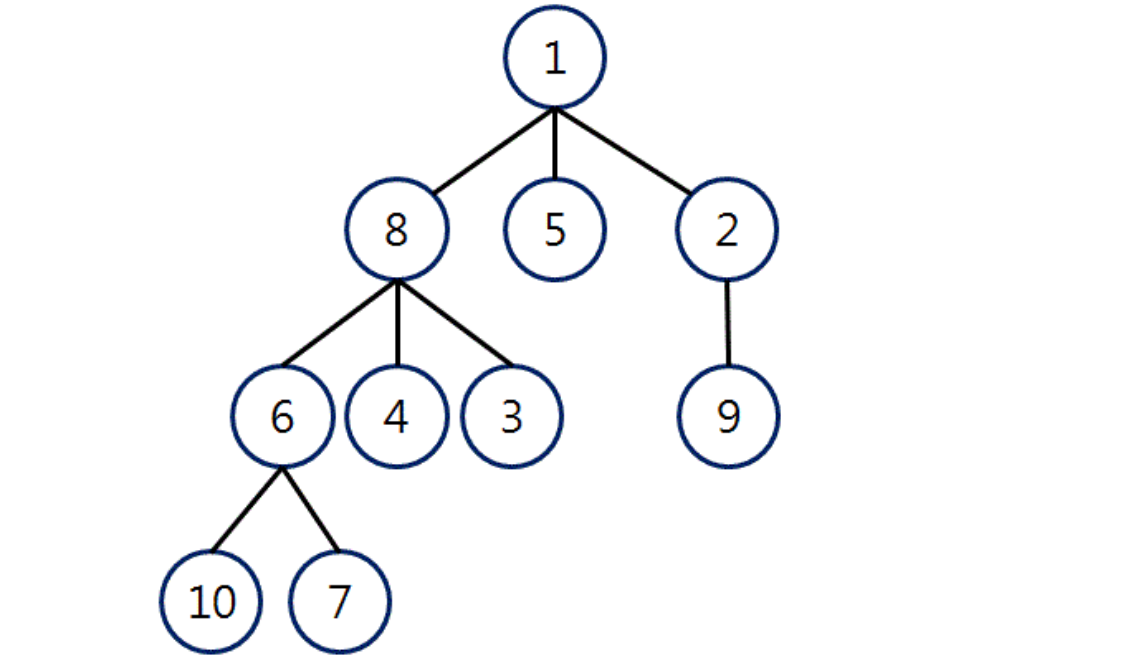
main () {
    khoitao ();
    cout<<endl;
    int solt=0;
    for (int u=1; u<=n; u++) {
        if (chuaxet[u]==TRUE) {
            solt++; //Tang solt (so lien thong) neu u chua xet
            cout<<endl<<"TPLT "<<solt<<": ";    //In ra so thanh phan lien thong
            DFS_TPLT (u);
        }
    }
}

```

## 2. Thuật toán duyệt đồ thị theo chiều rộng (Breadth-First Search - BFS)

**Định nghĩa**

BFS (Breadth-First Search) là tên gọi cho thuật toán tìm kiếm trong đồ thị dựa theo ưu tiên chiều rộng. Có nghĩa là, xuất phát từ một nút, ta bắt đầu duyệt tất cả các nhánh ở tầng liền kề nó, rồi từ các nhánh đó duyệt tiếp tầng liền kề tiếp theo, cứ như vậy cho tới hết vùng liên thông.




Trên hình là ví dụ của một đồ thị được duyệt từ nút 1 theo phương pháp BFS.

**Cách cài đặt BFS**

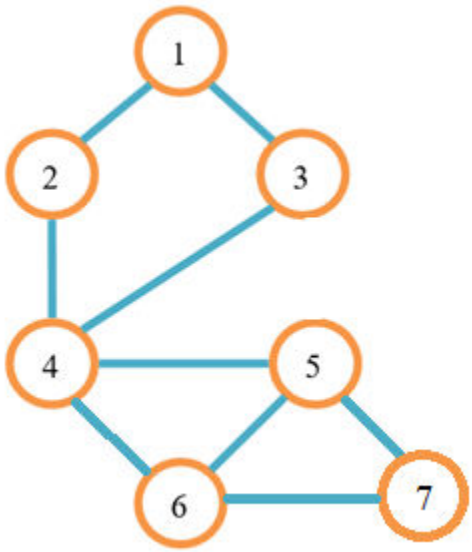
Đặc trưng của BFS để phân biệt với các thuật toán tìm kiếm đồ thị khác là việc sử dụng cấu trúc dữ liệu hàng đợi (queue).

Nguyên lý hoạt động ở đây sẽ là: ban đầu, nút đầu tiên của đồ thị sẽ được truyền vào hàng đợi. Từ đây trở đi, mỗi khi tới một nút thì nút đó sẽ được xuất khỏi hàng đợi, đồng thời chèn vào hàng đợi những nút liền kề với nút đó mà chưa được duyệt qua. Do cấu trúc hàng đợi là LIFO – một đầu vào, 1 đầu ra, nên những đỉnh ở lớp trước sẽ được lấy ra và những đỉnh mới duyệt tới sẽ được cho vào phía sau hàng đợi, đảm bảo luôn duyệt đồ thị từ những lớp gần với đỉnh xuất phát hơn trước. Thuật toán kết thúc khi hàng đợi rỗng.

***Bài toán cụ thể :*** Cho đồ thị vô hướng không trọng số như hình vẽ. Với điểm xuất phát là 1, yêu cầu duyệt qua các đỉnh sao cho mỗi đỉnh đúng 1 lần bằng thuật toán BFS.

***Áp dụng thuật toán BFS :*** Thuật toán tìm kiếm theo chiều rộng (*BFS*) là bắt đầu từ một đỉnh bất kì ***u*** nào đó duyệt đến các đỉnh v kề nó được một tập các đỉnh, tiếp tục chọn 1 đỉnh trong tập các đỉnh vừa lấy được đem đỉnh đó đi tìm các đỉnh chưa xét và đưa vào hàng đợi.

Ở đây nếu chọn đỉnh 1 đầu tiên: Ta có cách duyệt :




1 → 2,3 : được {2,3}

2 → 4 : được {3,4}

3 → Ø : được {4}

4 → 5,6 : được {5,6}

5 → 7 : được {5,6,7}

Danh sách đỉnh được duyệt lần lượt là :
1 , 2 , 3 , 4, 5 , 6, 7.

***Cài đặt thuật toán BFS :***

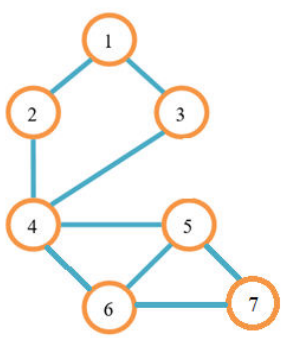
- Nhập dữ liệu biểu diễn đồ thị : cũng như DFS,ở đây chúng ta sẽ nhập vào các cặp cạnh (u, v) và biểu diễn đồ thị bằng danh sách kề.
- Cài đặt thuật toán BFS :
    - Đầu tiên khởi tạo hàng đợi chứa một đỉnh bắt đầu và đánh dấu đỉnh đó đã được xét, các đỉnh còn lại chưa xét
    - Mỗi bước thực hiện lấy một đỉnh ra khỏi hàng đợi và thực hiện duyệt đến các đỉnh kề với đỉnh đó mà chưa được xét, rồi thực hiện push lần lượt các đỉnh vào hàng đợi đồng thời đánh dấu đã xét.
    - Các bước lặp lại cho đến khi hàng đợi rỗng.

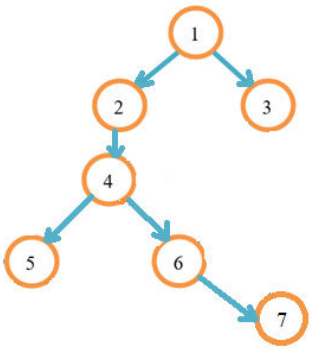
***Code C++ :*** BFS(1) sẽ cho phép duyệt đồ thị từ đỉnh 1.

```
18 void BFS(int s){
19     queue <int> Q ;
20     Q.push(s);  // khoi tao Q gom 1 dinh la dinh xuat phat
21     chuaxet[s] = false ;
22     while (!Q.empty()){
23         int u = Q.front(); Q.pop() ; // lay 1 dinh o dau Queue
24         cout<<u<<' ';
25         for (int i=0; i<ke[u].size(); i++){ // duyet tap cac dinh ke voi u
26             int v = ke[u][i];
27             if (chuaxet[v]){
28                 Q.push(v);      // neu gap v chua xet push(v) vao Queue
29                 chuaxet[v] = false ;// danh dau lai la da xet v
30             }
31         }
32     }
33 }
```

- ***Kiểm nghiệm thuật toán :***

| Bước | Queue | Đỉnh chọn | Đỉnh duyệt |
|---|---|---|---|
| 1 | 1 | 1 | 2 , 3 |
| 2 | 2 , 3 | 2 | 4 |
| 3 | 3 , 4 | 3 | Ø |
| 4 | 4 | 4 | 5 , 6 |
| 5 | 5 , 6 | 5 | 7 |
| 6 | 6 , 7 | 6 | Ø |
| 7 | 7 | 7 | Ø |
| 8 | Ø | Kết thúc | |




***<u>Ứng dụng BFS vào bài toán tìm đường đi ngắn nhất trên đồ thị không trọng số</u>:***

***Phát biểu bài toán :***

Cho đơn đồ thị vô hướng không trọng số gồm n đỉnh, m cạnh, và 2 đỉnh s, t thuộc tập đỉnh. Cho m cặp cạnh, tìm đường đi có tổng số cạnh đi qua là nhỏ nhất từ s → t

**<u>Nhận xét:</u>**

+ Khi thực hiện duyệt đồ thị theo chiều rộng, các đỉnh được duyệt sẽ phân thành các lớp, các đỉnh trong mỗi lớp đều có cùng khoảng cách tới đỉnh xuất phát.
+ Các đỉnh nằm ở lớp lớn hơn không thể có cách duyệt để nằm ở lớp bé hơn vì từ lớp trước sẽ duyệt được tất cả các đỉnh nằm ở lớp sau.
➔ Khoảng cách giữa đỉnh bắt đầu và đỉnh bất kỳ trong đồ thị khi sử dụng phương pháp duyệt BFS là đường đi có tổng số cạnh nhỏ nhất

+ Mỗi đỉnh được cập nhật vào hàng đợi theo 1 đỉnh khác ở lớp trước đó
➔ Ta có thể từ đỉnh đích dần lần vết về để in ra đường đi ngắn nhất từ s→t cho trước.

***Ý tưởng:***

- Thuật toán BFS bắt đầu từ đỉnh s khi duyệt sẽ cho ta đường đi nhỏ nhất đến đỉnh t.
- Sử dụng mảng truoc[1..n] : truoc[v] lưu đỉnh cha của v là u trong khi duyệt đồ thị đi từ u → v để phục vụ cho quá trình lần vết.

```
18 void BFS(int s){
19     queue <int> Q ;
20     Q.push(s);  // khoi tao Q gom 1 dinh la dinh xuat phat
21     chuaxet[s] = false ;
22     while (!Q.empty()){
23         int u = Q.front(); Q.pop() ; // lay 1 dinh o dau Queue
24         cout<<u<<' ';
25         for (int i=0; i<ke[u].size(); i++){ // duyet tap cac dinh ke voi u
26             int v = ke[u][i];
27             if (chuaxet[v]){
28                 Q.push(v);      // neu gap v chua xet push(v) vao Queue
29                 chuaxet[v] = false ;// danh dau lai la da xet v
30                 truoc[v] = u;  // di tu u-->v ==> truoc [v] = u;
31             }
32         }
33     }
```

***Truy vết:***

```
37 void truyvet(){
38     int u = t;  // truy vet tu dinh ket thuc
39     int p[nMax] ;
40     int d = 0;
41     while (u!=s){ // trong khi chua gap dinh bat dau
42         d++ ;
43         p[d] = u; // them u vao danh dach dinh tren duong di
44         u = truoc[u];
45     }
46     d++; p[d] = u;
47     cout<< "Do dai duong di nn tu"<< s<< "->"<<t <<":"<<d-1 <<endl;
48     for (int i=d; i>=1; i--){
49         cout<< p[i]<<' ';
50     }
51 }
```

***Code hoàn chỉnh :***

```cpp
#include <bits/stdc++.h>
using namespace std;
int const nMax = 1000006;
vector <int> ke[nMax] ;
int truoc[nMax];
int n , m ,s , t;
bool chuaxet[nMax] ;
void nhap(){
    cin>>n >> m >> s >> t; // so dinh va so canh + diem bat dau va ket thuc
    for (int i=1; i<= m ; i++){
        int u,v;
        cin>> u >> v;
        ke[u].push_back(v); // push dinh v vao danh sach ke voi dinh u
        ke[v].push_back(u); // push dinh u vao danh sach ke voi dinh v
    }

    for (int i=1; i<=n; i++) chuaxet[i] = true ;
}
void BFS(int s){
    queue <int> Q ;
    Q.push(s);  // khoi tao Q gom 1 dinh la dinh xuat phat
    chuaxet[s] = false ;
    while (!Q.empty()){
        int u = Q.front(); Q.pop() ; // lay 1 dinh o dau Queue
        cout<<u<<' ';
        for (int i=0; i<ke[u].size(); i++){ // duyet tap cac dinh ke voi u
            int v = ke[u][i];
            if (chuaxet[v]){
                Q.push(v);      // neu gap v chua xet push(v) vao Queue
                chuaxet[v] = false ;// danh dau lai la da xet v
                truoc[v] = u;  // di tu u-->v ==> truoc [v] = u;
            }
        }
    }
}

void truyvet(){
    int u = t;  // truy vet tu dinh ket thuc
    int p[nMax] ;
    int d = 0;
    while (u!=s){ // trong khi chua gap dinh bat dau
        d++ ;
        p[d] = u; // them u vao danh dach dinh tren duong di
        u = truoc[u];
    }
    d++; p[d] = u;
    cout<< "Do dai duong di nn tu"<< s<< "->"<<t <<":"<<d-1 <<endl;
    for (int i=d; i>=1; i--){
        cout<< p[i]<<' ';
    }
}


int main(){
    nhap();
    BFS(s); // duyet do thi bat dau tu dinh s
}
```

## III. Bài tập áp dụng

http://www.spoj.com/PTIT/problems/BCACM11D/

http://vn.spoj.com/problems/MTWALK/

http://vn.spoj.com/problems/VMUNCH/

http://vn.spoj.com/problems/QBBISHOP/

http://vn.spoj.com/problems/NKGUARD/

http://vn.spoj.com/problems/PBCWATER/

PROPTIT