Tên bài giảng
Sắp xếp, tìm kiếm nâng cao

Môn học: **Thuật toán và ứng dụng**

Chương: 3

Hệ: Đại học

Giảng viên: TS. Phạm Đình Phong

Email: phongpd@utc.edu.vn

Nội dung bài học

1. **Sắp xếp**
2. **Bảng băm**
3. **Tìm kiếm trên dữ liệu phân nhánh**
4. **Tìm kiếm tối ưu**

Sắp xếp

- Tại sao cần sắp xếp
  - Sắp xếp một danh sách các phần tử theo một thứ tự nào đó là một bài toán có ý nghĩa trong thực tiễn
  - Sắp xếp là một yêu cầu không thể thiếu trong thiết kế, phát triển các phần mềm ứng dụng
  - Nghiên cứu phương pháp sắp xếp là rất cần thiết

Sắp xếp

- Khái niệm
  - **Sắp xếp** là quá trình bố trí lại các phần tử trong một tập hợp theo một trình tự nào đó nhằm mục đích giúp quản lý và tìm kiếm các phần tử dễ dàng và nhanh chóng hơn
  - **Sắp xếp trong** là sự sắp xếp dữ liệu được tổ chức trong bộ nhớ trong của máy tính
  - **Sắp xếp ngoài** là sự sắp xếp được sử dụng khi số lượng phần tử cần sắp xếp lớn không thể lưu trữ trong bộ nhớ trong mà phải lưu trữ trên bộ nhớ ngoài

Sắp xếp

- Counting sort (sắp xếp đếm phân phối)
  - Counting sort là một thuật toán sắp xếp cực nhanh một mảng các phần tử mà mỗi phần tử là các số nguyên không âm hoặc là một danh sách các ký tự được ánh xạ về dạng số để sắp xếp theo bảng chữ cái
  - Counting sort không dựa vào so sánh

Sắp xếp

- Ý tưởng của Counting sort
  - Dựa vào các khóa trong một khoảng cụ thể
  - Nó làm việc dựa vào việc đếm số phần tử có giá trị khóa khác biệt (một kiểu băm). Sau đó thực hiện việc tính toán số học vị trí của từng phần tử trong chuỗi đầu ra

Sắp xếp

- Minh họa Counting sort
  - Để đơn giản, giả sử dữ liệu trong khoảng 0 đến 9
  - Dữ liệu đầu vào: 1, 4, 1, 2, 7, 5, 2

Bước 1: Đếm số lần xuất hiện của từng phần tử trong mảng cần sắp

Index:	0	1	2	3	4	5	6	7	8	9
Count:	0	2	2	0	1	1	0	1	0	0

Bước 2: Sửa mảng count sao cho chỉ số của mỗi phần tử lưu tổng số lần đếm trước đó → vị trí của chúng trong mảng đầu ra

Index:	0	1	2	3	4	5	6	7	8	9
Count:	0	2	4	4	5	6	6	7	7	7

Bước 3: Xuất từng phần tử trong chuỗi đầu vào tại ví trí là tổng số lần đếm trừ đi 1 và trừ số lần đếm đi 1.

Xử lý dữ liệu đầu vào: 1, 4, 1, 2, 7, 5, 2. Vị trí của 1 là (2-1) = 1 thì đặt số 1 vào chỉ số 1 trong mảng đầu ra. Gặp số 1 tiếp theo thì số lần đếm lúc này là 1 và giảm đi thành 0 và đặt số 1 vào chỉ số 0

Sắp xếp

- Counting sort
  - **Độ phức tạp thời gian**: $O(n+k)$, trong đó n là số phần tử trong dãy đầu vào và k là khoảng dữ liệu đầu vào
  - **Độ phức tạp không gian**: $O(n+k)$
  - **Nhược điểm**:
    - Độ phức tạp không gian lớn cho việc sắp xếp số lượng nhỏ các phần tử nhưng có khoảng giá trị (khoảng cách từ *số nhỏ nhất* và *số lớn nhất*) lớn vì nó đòi hỏi một lượng lớn không gian không được sử dụng
    - Điều gì xảy ra nếu các phần tử trong khoảng tử 1 đến n^2? Khi đó counting sort sẽ có độ phức tạp thời gian là $O(n^2)$

Sắp xếp

- Sắp xếp theo cơ số (Radix sort)
  - Một thuật toán sắp xếp không so sánh
  - Dựa trên nguyên tắc phân loại thư của bưu điện nên nó còn có tên là Postman sort
  - Bưu điện chuyển lượng lớn thư đến các địa phương khác nhau → tổ chức phân loại thư theo phân cấp
    - Các thư đến cùng một tỉnh/thành phố sẽ được đưa vào 1 lô
    - Các bưu điện tỉnh phân các thư theo lô cùng quận, huyện
    - Cứ như vậy, các thư sẽ đến được tay người nhận thư

Sắp xếp

- Sắp xếp theo cơ số (Radix sort)
  - Radix sort dựa trên ý tưởng nếu một danh sách đã được sắp xếp hoàn chỉnh thì từng phần tử cũng sẽ được sắp xếp hoàn chỉnh dựa trên giá trị của các phần tử đó
  - Để sắp xếp dãy $a_1, a_2, \ldots, a_n$ thuật toán Radix sort thực hiện như sau:
    - Giả sử mỗi phần tử a_i trong dãy $a_1, a_2, \ldots, a_n$ là một số nguyên có tối đa m chữ số
    - Phân loại các phần tử lần lượt theo các chữ số hàng đơn vị, hàng chục, hàng trăm, … tương tự việc phân loại thư theo tỉnh thành, quận/huyện, phường/xã, …

Sắp xếp

- Các bước thực hiện sắp xếp theo cơ số
  - **Bước 1**: $k = 0$; //(k = 0: hàng đơn vị; k = 1: hàng chục, …), k là chữ số hiện thời (tức là sắp xếp các số hàng đơn vị trước)
  - **Bước 2**: Khởi tạo 10 lô B_0, B_1, … , B_9 rỗng; //Tạo các lô chứa các loại chữ số khác nhau (từ 0 đến 9)
  - **Bước 3**:

    for (i = 0; i < n; i++) //Lặp từ phần tử a_0 đến a_{n-1}

    Đặt a_i vào lô B_t với t là chữ số thứ k của a_i;
  - **Bước 4**: Nối B_0, B_1, .. , B_9 lại (theo đúng trình tự) thành a
  - **Bước 5**:

    $k = k + 1$; //k = 0 → chuyển lên hàng chục, …

    Nếu $k < m$ thì trở lại **bước 2**.

    Ngược lại: Dừng

Sắp xếp

- Ví dụ sắp xếp theo cơ số
  - Dãy chưa được sắp:

    a = {10, 15, 1, 60, 5, 100, 25, 50}
  - Sắp xếp theo các số hàng đơn vị

    → Phân lô theo hàng đơn vị:

	50									
	100					25				
	60					5				
	10	1				15				
Cơ số (B_t)	0	1	2	3	4	5	6	7	8	9

→ a = {1**0**, 6**0**, 10**0**, 5**0**, **1**, 1**5**, **5**, 2**5**}

Sắp xếp

- Ví dụ sắp xếp theo cơ số
  - a = {1**0**, 6**0**, 10**0**, 5**0**, **1**, 1**5**, **5**, 2**5**}
  - Sắp xếp theo các số hàng chục

    → Phân lô theo hàng chục:

	005									
	001	15								
	100	10	25			50	60			
Cơ số (B_t)	0	1	2	3	4	5	6	7	8	9

→ a = {1**0**0, 1, 5, **1**0, **1**5, **2**5, **5**0, **6**0}

Sắp xếp

- Ví dụ sắp xếp theo cơ số
  - a = {1**0**0, 1, 5, **1**0, **1**5, **2**5, **5**0, **6**0}
  - Sắp xếp theo các số hàng trăm

    → Phân lô theo hàng trăm:

	60								
	50								
	25								
	15								
	10								
	5								
	1	100							
Cơ số (B_t)	0	1	2	3	4	5	6	7	8	9

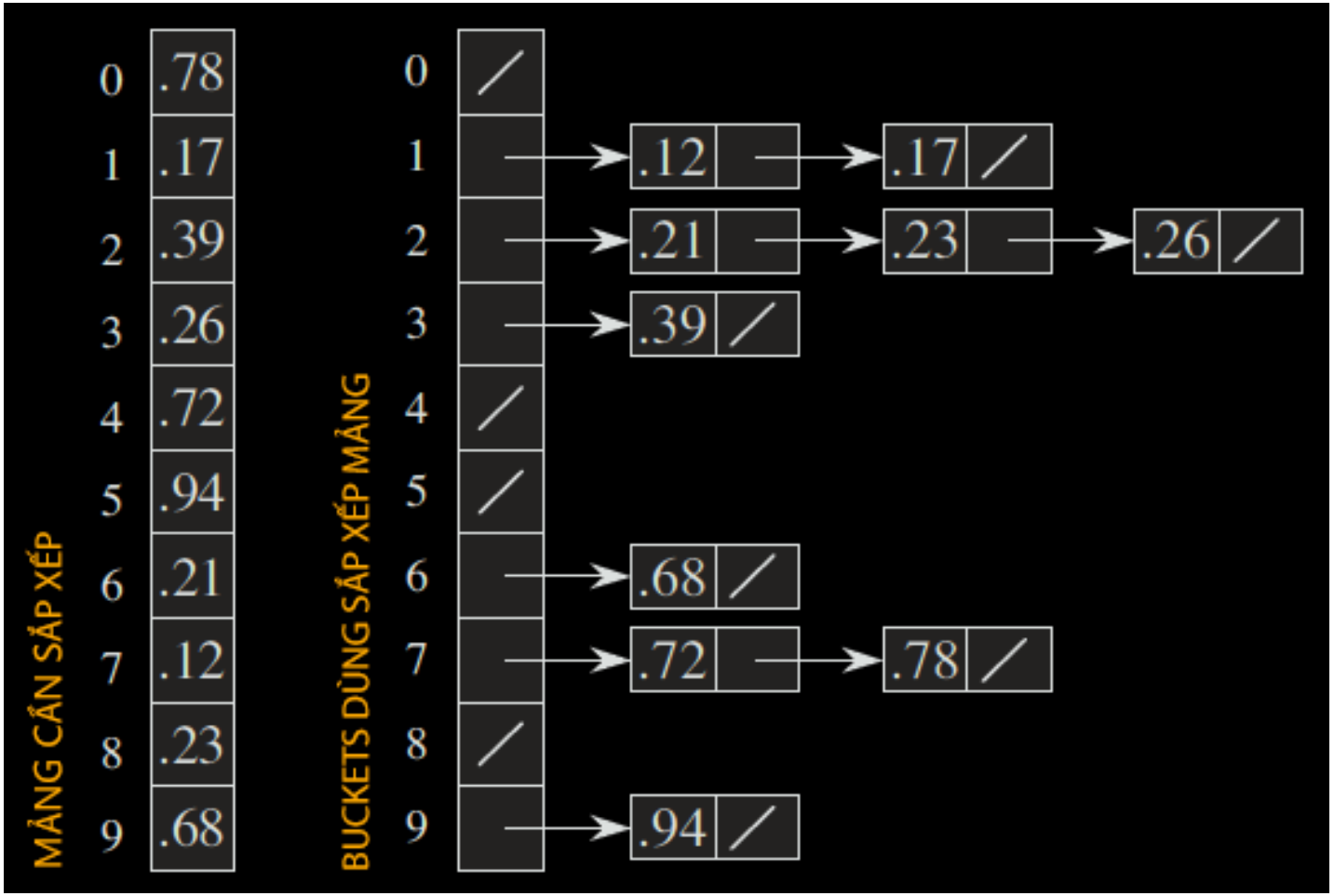
→ a = {1, 5, 10, 15, 25, 50, 60, **1**00}

Sắp xếp

- Bucket sort
  - Thuật toán bucket sort hữu ích khi sắp xếp các dãy các số thực được phân phối đều từ 0.0 đến 1.0
  - Thuật toán gồm các bước sau:
    - Tạo *n* xô rỗng
    - Đặt các phần tử của mảng cần sắp vào các xô (bucket) thích hợp
    - Sau khi đặt hết tất cả các phần tử vào trong các xô thì trong mỗi xô sắp xếp các phần tử trong xô theo thứ tự
    - Liên kết các xô lại trở thành mảng các phần tử đã được sắp xếp theo thứ tự

Sắp xếp

- Bucket sort



Sắp xếp

- Bài tập
  - Cho mảng số nguyên. Hãy sắp xếp chúng theo nguyên tắc.

    a đứng trước b nếu tổng các chữ số của a nhỏ hơn b

    Nếu hai số có tổng các chữ số bằng nhau, số nào nhỏ hơn sẽ đứng trước. Ví dụ 4 và 13 có tổng chữ số bằng nhau, nhưng 4 < 13 nên 4 sẽ đứng trước 13 trong mảng kết quả
  - Ví dụ: a = {13, 20, 7, 4} thì kết quả là: [20, 4, 13, 7]

Sắp xếp

- Bài tập
  - Hùng có một mảng gồm N số nguyên nhưng anh ấy không thích mảng số này. Anh ta muốn tạo nó thành một mảng đẹp (beautiful) và một mảng A_1, A_2, A3, . . ., A_N là đẹp nếu $A_1 > A_N$. Để tạo mảng này thành mảng đẹp, anh ta đổi giá trị của hai số bất kỳ trong mảng. Anh ta có thể thực hiện thao tác này nhiều lần trên các cặp số nguyên kề nhau trong mảng A. Hãy tìm số cách mà Hùng có thể tạo mảng A thành mảng đẹp. Hai cách được xem là giống nhau nếu sau khi tạo mảng đẹp thì hai số A_1 và A_N là giống nhau.
  - Ví dụ: A = {1, 4, 3, 2, 5}

    Tổng số cách là (5,1), (4,1), (3,1), (2,1), (5,2), (4,2), (3,2), (5,3), (4,3), (5,4) với số đầu là A_1 và số thứ hai là A_N.

Sắp xếp

- Bài tập
  - Bạn được cho một mảng A kích thước N và một số nguyên K. Mảng này gồm N số nguyên có giá trị từ 1 đến 107. Mỗi phần tử a[i] của mảng này có **trọng số đặc biệt** được tính: a[i] % K. Bạn cần sắp xếp mảng này theo thứ tự giảm dần của các **trọng số đặc biệt**, tức là phần tử có trọng số đặc biệt lớn nhất thì đứng ở đầu dãy, tiếp đến là phần tử có trọng số đặc biệt lớn thứ hai, … Trong trường hợp hai phần tử có cùng trọng số đặc biệt thì phần tử có giá trị nhỏ hơn sẽ đứng trước

Sắp xếp ngoài

- Đặt vấn đề
  - Vì sao phải xây dựng thuật toán sắp xếp ngoài hay sắp xếp trên file?
    - Khi dữ liệu cần sắp xếp quá lớn không thể tổ chức sắp xếp trong (bộ nhớ trong) bằng các thuật toán sắp xếp phổ dụng
    - Thao tác cơ bản trong sắp xếp thông thường là hoán vị → Thực hiện hoán vị trên cùng 1 file dẫn đến tần suất thao tác trên một vùng đĩa quá lớn → Không an toàn
    - Sắp xếp ngoài không thích hợp với hoán vị

Sắp xếp ngoài

- Mục tiêu
  - Đặc trưng của sắp xếp trên file là bài toán trộn
  - Thuật toán tìm kiếm cơ bản: Tuần tự, nhị phân
  - Thuật toán sắp xếp: Trộn trực tiếp, trộn tự nhiên, trộn n-đường cân bằng, trộn đa pha
  - Đánh giá thuật toán

Sắp xếp ngoài

- Bài toán trộn
  - **Đầu vào**: Cho trước 2 (hay nhiều) dãy

    $A[0], A[1], A[2], \ldots, A[N-1]$

    $B[0], B[1], B[2], \ldots, B[M-1]$
  - **Đầu ra**: Kết hợp các dãy đã cho thành một dãy mới *C* được sắp

Sắp xếp ngoài

- Thuật toán 1 – Trộn từng phần tử

```
int currA = 0, currB = 0, currC = 0;
A[N] = MAX_VALUE; // phần tử lính canh
B[M] = MAX_VALUE; // phần tử lính canh
for (int currC=0; currC < N + M; currC++)
    if (A[currA] < B[currB]) {
        C[currC] = A[currA];
        currA++;
    }
    else {
        C[currC] = B[currB];
        currB++;
    }
```

Sắp xếp ngoài

- Thuật toán 2: Trộn dãy phần tử được sắp

```
int currA = 0, currB = 0, currC = 0;
while (currA < N && currB < M) {
        if (A[currA] < B[currB])
                C[currC] = A[currA++];
        else C[currC] = B[currB++];
        currC++;
}
// Xét phần tử còn lại của dãy A
for ( ; currA < N; currA++) {
        C[currC] = A[currA];
        currC++;
}
// Xét phần tử còn lại của dãy B
for ( ; currB < M; currB++) {
        C[currC] = B[currB];
        currC++;
}
```

Sắp xếp ngoài

- Phương pháp thông thường
  - Giả sử tập dữ liệu cần sắp xếp trong file f0 → đọc ra *a* số đầu tiên, sắp xếp rồi ghi vào file f1
  - Đọc tiếp *a* số tiếp theo, sắp xếp rồi tiến hành trộn a số được sắp đó với dữ liệu đã được sắp trong file f1 → ghi 2**a* số đã được sắp vào file f2
  - Tiếp tục đọc *a* số nữa từ file f0, sắp xếp rồi trộn với 2**a* số trong file f2 được 3**a* số được sắp rồi ghi vào file f1. Vai trò của f1 và f2 tráo đổi cho nhau
  - Đến đây ta đã thấy quy luật → lặp lại quá trình cho đến khi đọc hết file f0. Kết quả cuối cùng sẽ nằm ở file f1 hoặc f2 tùy vào vòng lặp dừng ở đâu

Sắp xếp ngoài

- Phương pháp trộn Run - Thuật toán
  - Thuật toán sắp xếp tập tin bằng phương pháp trộn Run có thể tóm lược như sau:
  - **Input**: f0 là tập tin cần sắp thứ tự
  - **Ouput**: f0 là tập tin đã được sắp thứ tự
  - Gọi f1, f2 là 2 tập tin trộn
  - Trong đó f0, f1, f2 có thể là các tập tin văn bản thường (text file) hoặc các tập tin nhị phân

Sắp xếp ngoài

- Phương pháp trộn Run – Minh họa
  - **Bước 1**:
    - Giả sử các phần tử trên f0 là:

    24 12 67 33 58 42 11 34 29 31

    - Khởi tạo f1, f2 rỗng
    - Thực hiện phân bố *m* = 1 phần tử từ f0 lần lượt vào f1, f2:

    f1: **24 67 58 11 29**

    f2: **12 33 42 34 31**

    - Trộn f1,f2 vào f0

    f0: **12 24 33 67 42 58 11 34 29 31**

Sắp xếp ngoài

- Phương pháp trộn Run – Minh họa
  - **Bước 2**:
    - Phân bố m = 2 *m = 2 phần tử từ f0 vào f1, f2
    - f0: 12 24 33 67 42 58 11 34 29 31
    - f1: 12 24 42 58 29 31
    - f2: 33 67 11 34
    - Trộn f1, f2 thành f0:
    - f0: 12 24 33 6711 34 42 58 29 31

Sắp xếp ngoài

- Phương pháp trộn Run – Minh họa
  - **Bước 3**:
    - Tương tự bước 2, phân bố m = 2 * m = 4 phần tử từ f0 vào f1, f2

      f0: 12 24 33 67 11 34 42 58 29 31

      f1: 12 24 33 67 29 31

      f2: 11 34 42 58
    - Trộn f1, f2 thành f0:

      f0: 11 12 24 33 34 42 58 67 29 31

Sắp xếp ngoài

- Phương pháp trộn Run – Minh họa
  - **Bước 4**:
    - Phân bố m = 2 * m = 8 phần tử lần lượt từ f0 vào f1, f2

      f1: 11 12 24 33 34 42 58 67

      f2: 29 31
    - Trộn f1, f2 thành f0:

      f0: 11 12 24 29 31 33 34 42 58 67
  - **Bước 5**:
    - Lặp lại tương tự các bước trên cho tới khi chiều dài *m* của run cần phân bố lớn hơn chiều dài *N* của f0 thì dừng

Sắp xếp ngoài

- Phương pháp trộn Run – Cài đặt

m = 1
while (m < số phần tử của f0)
{
    Chia [Distribute] *m* phần tử của f0 lần lượt cho f1 và f2
    Trộn [Merge] f1 và f2 lần lượt vào f0
    m = *m* * 2
}

Sắp xếp ngoài

- Phương pháp trộn Run – Đánh giá
  - Đánh giá
    - Cần ít nhất *N* không gian trống trên đĩa để hoạt động
    - Số bước $\log_2 N$ (vì mỗi lần xử lý 1 dãy tăng gấp 2)
    - **Mỗi bước**:
      - Phân phối (Distribute): Copy *N* lần
      - Trộn (Merge): Copy *N* lần, so sánh *N*/2 lần
    - **Tổng cộng**:
      - Copy: $2N * \log_2 N$
      - So sánh: $N/2 * \log_2 N$
    - **Hạn chế**:
      - Không tận dụng được dữ liệu đã được sắp bộ phận
      - Độ dài dãy con xử lý ở bước k <= 2^k

Sắp xếp ngoài

- Khái niệm Run
  - Run là một dãy liên tiếp các phần tử được sắp thứ tự
  - Ví dụ: 2 4 7 12 55 là một Run
  - Chiều dài Run chính là số phần tử trong Run
  - Chẳng hạn Run ở ví dụ trên có chiều dài là 5

Sắp xếp ngoài

- Phương pháp trộn tự nhiên – Đặc điểm
  - Trong phương pháp trộn ở trên, thuật toán chưa tận dụng được chiều dài cực đại của các Run trước khi phân bố → chưa tối ưu
  - Đặc điểm của phương pháp trộn tự nhiên là tận dụng chiều dài "tự nhiên" của các Run ban đầu

    → Nghĩa là thực hiện việc trộn các Run có độ dài cực đại với nhau cho tới khi dãy chỉ còn một Run duy nhất → dãy đã được sắp

Sắp xếp ngoài

- Phương pháp trộn tự nhiên – Thuật toán

```
while (số Run của f0 > 1)
{
    Phân bố f0 vào f1, f2 theo các Run tự nhiên
    Trộn các Run của f1, f2 vào f0
}
```

  - **Phân bố**: Chia xoay vòng dữ liệu của file f0 cho f1 và f2, mỗi lần 1 run cho đến khi file f0 hết
  - **Trộn**: Trộn từng cặp run của f1 và f2 tạo thành run mới trên f0

Sắp xếp ngoài

- Phương pháp trộn tự nhiên – Minh họa
  - **Ví dụ**: f0: 1 2 9 8 7 6 5
  - **Bước 1**:

    f1: 1 2 9 7 5

    f2: 8 6

    f0: 1 2 8 9 6 7 5
  - **Bước 2**:

    f1: 1 2 8 9 5

    f2: 6 7

    f0: 1 2 6 7 8 9 5

Sắp xếp ngoài

- Phương pháp trộn tự nhiên – Minh họa
  - **Bước 3**: f0: 1 2 6 7 8 9 5

    f1: 1 2 6 7 8 9

    f2: 5

    f0: 1 2 5 6 7 8 9
  - Bước 4: Dừng vì f0 chỉ có một Run

Sắp xếp ngoài

- Phương pháp trộn đa lối cân bằng
  - Thuật toán sắp xếp ngoài cần 2 giai đoạn: Phân phối và trộn
    - Giai đoạn nào làm thay đổi thứ tự?
    - Chi phí cho giai đoạn phân phối?
  - Kết luận
    - Thay vì thực hiện 2 giai đoạn, ta chỉ cần thực hiện 01 giai đoạn trộn

      → Tiết kiệm một nửa chi phí Copy

      → Cần số lượng file trung gian gấp đôi

Sắp xếp ngoài

- Phương pháp trộn đa lối cân bằng
  - Chi phí sắp xếp ngoại tỉ lệ với số bước thực hiện:
    - Nếu mỗi bước cần *N* thao tác copy
    - Nếu dùng 2 file trung gian cần $\log_2 N$ bước → cần $N*\log_2 N$ thao tác copy
    - Để giảm số bước → Phân bố số Run nhiều hơn 2 file trung gian
    - Nếu dùng *n* file trung gian: cần $\log_n N$ bước → cần $N*\log_n N$ thao tác copy
  - Kết luận
    - Dùng nhiều file trung gian để giảm số bước
    - Tiết kiệm thao tác copy bằng cách thực hiện 1 giai đoạn
    - Sử dụng 2**n* file trung gian: *n* file nguồn và *n* file đích

Sắp xếp ngoài

- Phương pháp trộn đa lối cân bằng
  - Gọi tập nguồn *S* = {f1, f2, …, fn}

    Tập đích *D* = {g1, g2, …, gn}
  - **Bước 1**: Chia xoay vòng dữ liệu của file f0 cho các file thuộc tập nguồn, mỗi lần 1 Run cho tới khi f0 hết
  - **Bước 2**: Trộn từng bộ Run của các file thuộc tập nguồn *S*, tạo thành Run mới, mỗi lần ghi lên các file thuộc tập đích *D*
  - Nếu số Run trên các file của *D* > 1 thì
    - Hoán vị vai trò tập nguồn (*S*) và tập đích (*D*)
    - Quay lại Bước 2

    Ngược lại kết thúc thuật toán

Sắp xếp ngoài

- Phương pháp trộn đa lối cân bằng – Minh họa
  - Ví dụ: Cho dãy số sau

    3 5 2 7 12 8 4 15 20 1 2 8 23 7 21 27
  - Nhập

    f0: 3 5 2 7 12 8 4 15 20 1 2 8 23 7 21 27
  - Xuất

    f0:1 2 2 3 4 5 7 7 8 8 12 15 20 21 23 27

Sắp xếp ngoài

- Phương pháp trộn đa lối cân bằng – Minh họa
  - Các bước tiến hành: Chọn 6 file

    3 5 2 7 12 8 4 15 20 1 2 8 23 7 21 27
  - Bước 0: đặt *n* = 3
  - Bước 1: Phân phối các run luân phiên vào f1, f2, f3

    f1: 3 5 4 15 20

    f2: 2 7 12 1 2 8 23

    f3: 8 7 21 27

Sắp xếp ngoài

- Phương pháp trộn đa lối cân bằng – Minh họa
  - **Bước 2**: đặt n = 3

    Trộn các run của f1, f2, f3 và luân phiên phân phối vào các file g1, g2, g3

    g1: 2 3 5 7 8 12

    g2: 1 2 4 7 8 15 20 21 23 27

    g3:

    Do số run sau khi trộn >1 nên tiếp tục trộn run từ g1, g2, g3 vào ngược trở lại f1, f2, f3

    f1: 1 2 2 3 4 5 7 7 8 8 12 15 20 21 23 27

    f2:

    f3:

    Do số run trộn = 1 nên kết thúc thuật toán

Sắp xếp ngoài

- Phương pháp trộn đa lối cân bằng – Minh họa
  - Đầu vào f0: U Q N M K I H F D C B, *n* = 3

// Phân phối (lần 1)

f1: U M H C

f2: Q K F B

f3: N I D

// Trộn (lần 1)

g1: N Q U B C

g2: I K M

g3: D F H

// Trộn (lần 2)

f1: D F H I K M N Q U

f2: B C

f3: NULL

// Trộn (lần 3)

g1: B C D F H I K M N Q U

g2: NULL

g3: NULL

Sắp xếp ngoài

- Phương pháp trộn đa lối cân bằng – Thuật toán
  - Các ký hiệu:
    - fInput: file dữ liệu gốc cần sắp xếp
    - N: số phần tử trên file fInput
    - n: số file trung gian trên mỗi tập nguồn/đích
    - S: tập các file nguồn, D: tập các file đích
    - S_{dd}: tập các file nguồn đang còn run dở dang
    - S_{tr}: tập các file nguồn chưa hết (!EOF), còn có thể tham gia vào quá trình trộn
    - "Lượt": là 1 quá trình trộn run từ nguồn đến đích, một "lượt" kết thúc khi mỗi file đích (trong tập D) nhận được 1 run
    - D_{run}: tập các file đích đã nhận được run trong "lượt" hiện
    - $S - S_{tr}$: tập các file nguồn đã hết (EOF)
    - $S_{tr} - S_{dd}$: tập các file nguồn chưa hết (!EOF), nhưng đã kết thúc run hiện tại
    - $D - D_{run}$: tập các file đích chưa nhận được run trong "lượt" hiện hành

Sắp xếp ngoài

- Phương pháp trộn đa lối cân bằng – Thuật toán
  - **Bước 1**:

S = {f1, f2, …, fn}

D = {g1, g2, …, gn}

// Chia xoay vòng dữ liệu của fInput cho các file thuộc tập nguồn S

i = 1;

while (!feof(fInput)) {

    Copy_1_Run(fInput, fi);

    i = (i % n) + 1;

}

S_{tr} = S;

D_{run} = {};

nDemRun = 0;

Sắp xếp ngoài

- Phương pháp trộn đa lối cân bằng – Thuật toán
  - **Bước 2**:

a. $S_{dd} = S_{tr}$
b. Gọi $d_{hh} \in D - D_{run}$ là file đích hiện hành (sẽ được nhận run)
c. Đọc các phần tử x_{fi}, fi $\in S_{dd}$
d. Gọi x_{f0}= MIN {x_{fi}, fi $\in S_{dd}$}
e. Copy x_{f0} lên d_{hh}
f. **Nếu (file f0 hết) thì** {
    $S_{tr} = S_{tr} - \{f0\}$; $S_{dd} = S_{dd} - \{f0\}$
    Nếu (S_{tr} == {}) thì { // Xong quá trình trộn N đến D
        nDemRun++; Goto [Bước 3]
    }
    ngược lại Nếu (S_{dd}<> {}) thì Goto [Bước 2.d]
    ngược lại { // S_{dd}=={}: hết bộ run hiện hành
        nDemRun++; $D_{run} = D_{run} + \{d_{hh}\}$;
        Nếu (D_{run} == D) thì D_{run}= {}; // Xong 1 "lượt"
        Goto [Bước 2.a]
    }

Sắp xếp ngoài

- Phương pháp trộn đa lối cân bằng – Thuật toán
  - **Bước 2**:

ngược lại { // File f0 chưa hết
    Nếu (!EOR(f0)) thì {
        Đọc phần tử kế x_{f0} từ file f0;
        Goto [Bước 2.d]
    }
    ngược lại { // Hết run hiện hành trên f0
        $S_{dd} = S_{dd} - \{f0\}$
        Nếu ($S_{dd} <> \{\}$) thì Goto [Bước 2.d]
        ngược lại { // $S_{dd} == \{\}$: hết bộ run hiện hành
        nDemRun++;
        $D_{run} = D_{run} + \{d_{hh}\}$;
        Nếu ($D_{run} == D$) thì $D_{run} = \{\}$; // Xong 1 "lượt"
        Goto [Bước 2.a]
    }
} // Kết thúc "file f0 chưa hết"

Sắp xếp ngoài

- Phương pháp trộn đa lối cân bằng – Thuật toán
  - **Bước 3**:

    Nếu (nDemRun == 1) thì Kết thúc thuật toán

    ngược lại {

        Nếu (nDemRun < n) thì $S_{tr} = D_{run}$; // Không đủ n run

        ngược lại $S_{tr} = D$;

        D_{run}= {};

        nDemRun = 0;

        Hoán vị tập S và D

        Goto [Bước 2]

    }

Sắp xếp ngoài

- Phương pháp trộn đa pha
  - Với phương pháp trộn đa lối cân bằng, các tập tin chưa được sử dụng một cách có hiệu quả bởi vì trong cùng một lần duyệt thì phân nửa số tập tin luôn luôn giữ vai trò trộn (nguồn) và phân nửa số tập tin luôn luôn giữ vai trò phân phối (đích)
  - Cải tiến: Thay đổi vai trò của các tập tin trong cùng một lần duyệt → phương pháp trộn đa pha

Sắp xếp ngoài

- Phương pháp trộn đa pha
  - Xét ví dụ sau với 3 tập tin f1, f2, f3
  - **Bước 1**: Phân phối luân phiên các Run ban đầu của f0 vào f1 và f2
  - **Bước 2**: Trộn các run của f1, f2 vào f3. Thuật kết thúc nếu f3 chỉ có một Run
  - **Bước 3**: Chép nửa số run của f3 vào f1
  - **Bước 4**: Trộn các Run của f1 và f3 vào f2. Giải thuật kết thúc nếu f2 chỉ có một Run
  - **Bước 5**: Chép nửa số Run của f2 vào f1. Lặp lại bước 2
  - Phương pháp này còn có nhược điểm là mất thời gian sao chép nửa số Run của tập tin này vào tập tin kia. Việc sao chép này có thể loại bỏ nếu ta bắt đầu với **fn** Run của tập tin f1 và **fn-1** Run của tập tin f2, với **fn** và **fn-1** là các số liên tiếp trong **dãy Fibonaci**

Sắp xếp ngoài

- Bài tập
  - Cài đặt các thuật toán sắp xếp ngoài bằng C++ và Python

Bảng băm – Hash table

- Khái niệm băm (hashing)
  - Băm (Hashing) là một kỹ thuật dùng để xác định duy nhất một đối tượng cụ thể từ một nhóm các đối tượng tương tự nó
  - Ví dụ về hashing trong thực tế
    - Mỗi sinh viên trong một trường đại học được giao cho một số định danh (ID) và ta có thể tìm kiếm các thông tin liên quan đến sinh viên từ số định danh này
    - Mỗi cuốn sách trong thư viện cũng được gán cho một định danh (ID) duy nhất và dựa vào số định danh này ta có thể biết được vị trí chính xác của cuốn sách đó trong thư viện hoặc các người dùng đã mượn nó
  - Trong các ví dụ trên, sách và sinh viên đã được mã hóa với một ID duy nhất, kỹ thuật đó gọi là băm

Bảng băm – Hash table

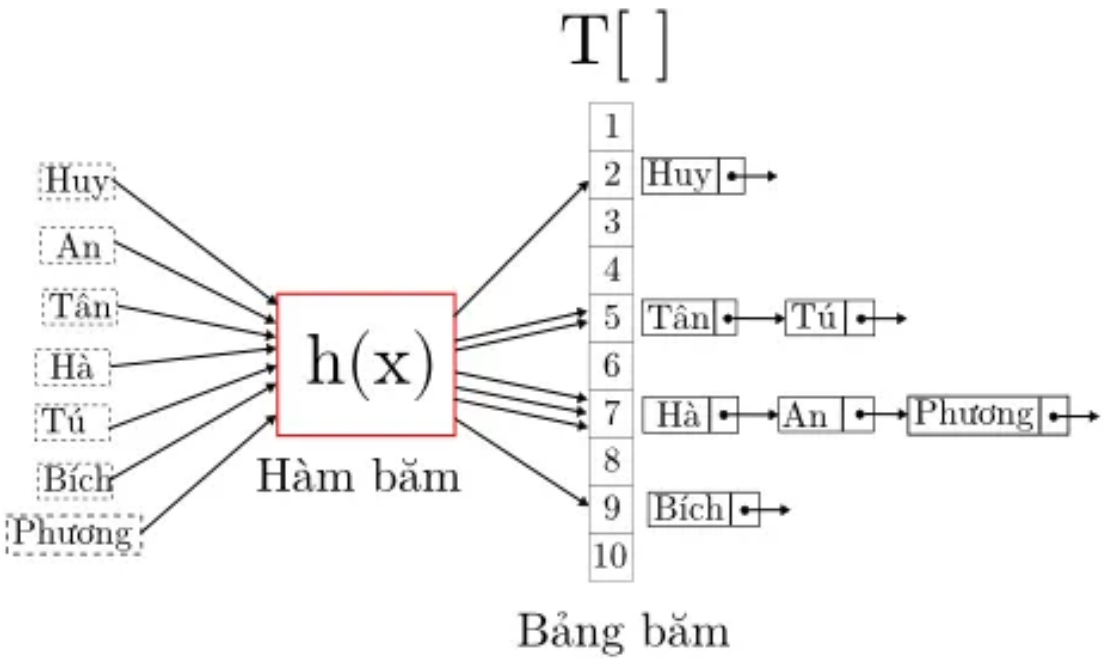
- Khái niệm băm (hashing)
  - Có thể định nghĩa băm là kỹ thuật biến đổi một khóa (key) trở thành một số nguyên bằng cách dùng các công thức toán học
  - Công thức toán học này được thực hiện trong một hàm gọi là **hàm băm** (hash function)
  - Một khóa khi qua xử lý của hàm băm sẽ sinh ra một số nguyên duy nhất được gọi là **mã băm** (hash code)

Bảng băm – Hash table

- Kỹ thuật băm
  - Giả sử có một đối tượng được gán một số định danh (ID) để dễ quản lý. Khi gán cho đối tượng một ID thì một cặp key/value được hình thành với key là ID và value chính là đối tượng được gán ID đó
  - Để thực hiện việc trên, có thể dùng mảng và khi này key chính là chỉ số của mảng, nơi lưu trữ đối tượng (value)
  - Trong trường hợp key quá lớn thì việc dùng key trực tiếp làm chỉ số của mảng không còn hiệu quả nữa và khi đó kỹ thuật băm (hashing) được áp dụng

Bảng băm – Hash table

- Kỹ thuật băm
  - Sau khi băm, cặp key/value (key đã dược chuyển đổi) được chứa trong một cấu trúc dữ liệu là bảng băm (hash table)
  - Bằng cách sử dụng key trong bảng băm, ta có thể truy cập các phần tử trong bảng này trong thời gian O(1)



Bảng băm – Hash table

- Kỹ thuật băm
  - Có thể được thực hiện trong hai bước
    - Chuyển đổi khóa (key) thành số nguyên (khóa nhỏ hơn) dùng hàm băm. Khóa sau khi chuyển đổi có thể được dùng như chỉ số để chứa phần tử ban đầu nằm trong bảng băm
    - Phần tử được chứa trong bảng băm có thể được truy cập nhanh chóng qua khóa đã được chuyển đổi

```
hash_code = hash_function(key)
index = hash_code % array_size
```

    - Với cách chuyển đổi này, hash_code được tính độc lập với kích thước của mảng, sau đó giá trị của nó được chuyển đổi về chỉ số của mảng là index có giá trị từ 0 tới array_size - 1

Bảng băm – Hash table

- Hàm băm (hash function)
  - Là một hàm bất kỳ có thể được sử dụng để ánh xạ một bộ dữ liệu có kích thước tùy ý tới một bộ dữ liệu có kích thước cố định nằm trong bảng băm.
  - Giá trị trả về bởi hàm băm được gọi là mã băm (hash code hoặc hash value)
  - Điều kiện để có một hàm băm tốt
    - **Dễ tính toán**: Phải dễ tính toán và bản thân nó không phải là một thuật toán
    - **Phân phối đồng đều**: Cần phải phân phối đồng đều trên bảng băm, không xảy ra việc tập trung thành các cụm
    - **Ít xung đột**: Sự xung đột (collision) xảy ra khi có các cặp phần tử khác nhau được ánh xạ tới cùng một mã băm

Bảng băm – Hash table

- Xung đột và xử lý xung đột
  - Sự xung đột trong bảng băm gây ảnh hưởng lớn tới hiệu năng của loại cấu trúc dữ liệu này → cần sử dụng các kỹ thuật để giảm xung đột trong bảng
  - Ví dụ: ta có các chuỗi ký tự sau

    {"abcdef", "bcdefa", "cdefab" , "defabc"}

    → dùng kỹ thuật băm để lưu các chuỗi ký tự trên trong một bảng băm
  - Để tính toán các chỉ số cho việc lưu trữ các chuỗi, ta xây dựng hàm băm với các cách như sau

Bảng băm – Hash table

- Xung đột và xử lý xung đột
  - **Cách thứ nhất**
    - Chỉ số cho mỗi chuỗi bằng tổng giá trị mã ASCII của mỗi phần ký tự trong chuỗi
    - Gọi giá trị mã ASCII của một ký tự *x* là asc(x), istr(s) là tổng giá trị mã ASCII của chuỗi s, ta có

istr(abcdef) = asc(a) + asc(b) + asc(c) + asc(d) + asc(e) + asc(f) = 599
istr(bcdefa) = asc(b) + asc(c) + asc(d) + asc(e) + asc(f) + asc(a) = 599
istr(cdefab) = asc(c) + asc(d) + asc(e) + asc(f) + asc(a) + asc(b) = 599
istr(defabc) = asc(d) + asc(e) + asc(f) + asc(a) + asc(b) + asc(c) = 599
Với asc(a) = 97, asc(b) = 98, asc(c) = 99, asc(d) = 100, asc(e) = 101, asc(f) = 102

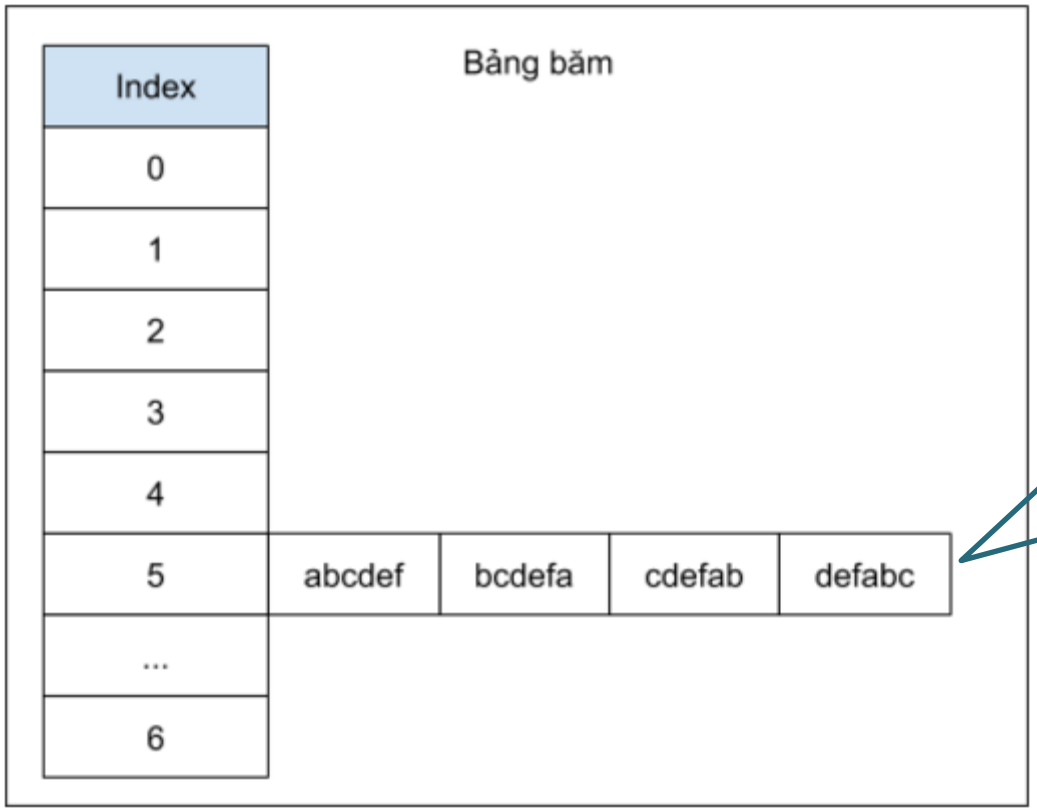
    - Chỉ số được tính toán bằng cách chia lấy phần dư istr(s) cho số ký tự có trong chuỗi s

Bảng băm – Hash table

- Xung đột và xử lý xung đột
  - **Cách thứ nhất**
    - Gọi chỉ số của chuỗi s là idx(s) ta có:

      idx(abcdef) = idx(bcdefa) = idx(cdefab) = idx(defabc)

      = 599 % 6 = 5 → ánh xạ tới cùng một chỉ số trên bảng băm



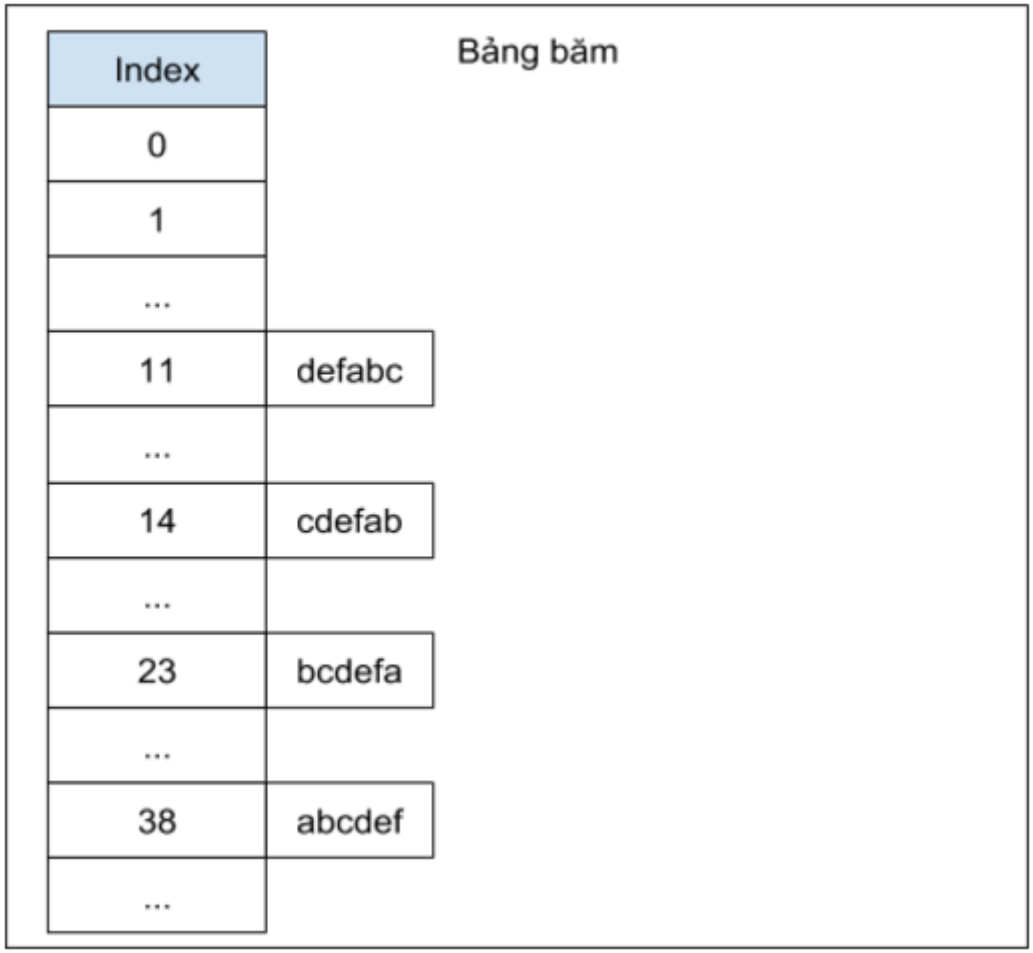
Bảng băm – Hash table

- Xung đột và xử lý xung đột
  - **Cách thứ hai**
    - Chỉ số của một chuỗi sẽ bằng tổng mã ASCII của từng ký tự nhân với thứ tự sắp xếp của ký tự đó trong chuỗi

Chuỗi	Tính toán trong hàm băm	Chỉ số
abcdef	(97*1 + 98*2 + 99*3 + 100*4 + 101*5 + 102*6) % 2069	38
bcdefa	(98*1 + 99*2 + 100*3 + 101*4 + 102*5 + 97*6) % 2069	23
cdefab	(99*1 + 100*2 + 101*3 + 102*4 + 97*5 + 98*6) % 2069	14
defabc	(100*1 + 101*2 + 102*3 + 97*4 + 98*5 + 99*6) % 2069	11

Bảng băm – Hash table

- Xung đột và xử lý xung đột
  - **Cách thứ hai**
    - Ta có bảng băm tối ưu hơn



Bảng băm – Hash table

- Bảng băm - Hash table
  - Bảng băm là một loại cấu trúc dữ liệu được dùng để chứa cặp key/value
  - Chỉ số được tính toán bởi hàm băm → giúp cho việc chèn hoặc tìm kiếm dữ liệu được dễ dàng hơn
  - Với một bảng băm có một hàm băm được thực hiện tốt thì trong trường hợp lý tưởng, việc tìm kiếm các phần tử trong bảng có thời gian là O(1)

Bảng băm – Hash table

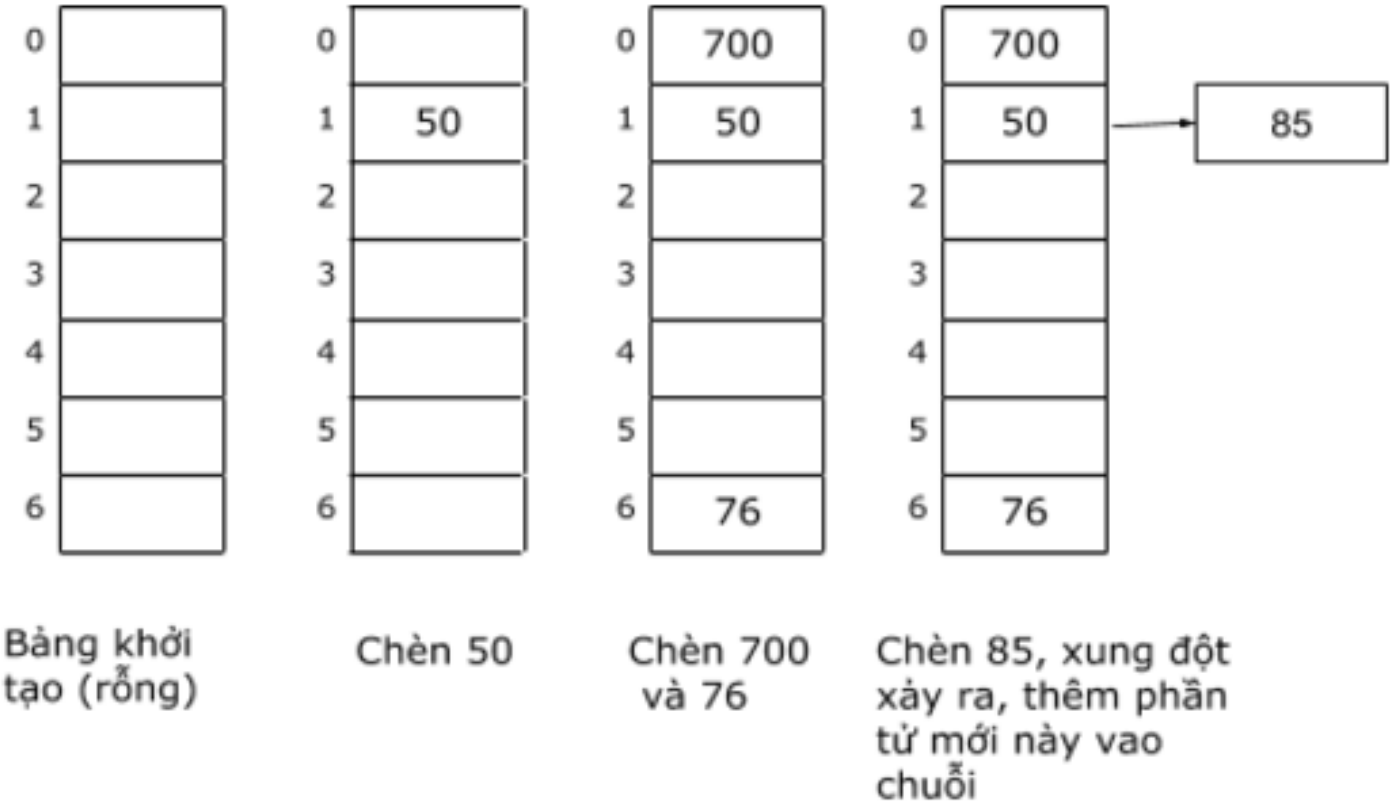
- Bảng băm - Hash table
  - Ví dụ: Đếm số lần xuất hiện của các ký tự trong chuỗi: “ababcd”
  - **Cách thông thường:** duyệt qua tất cả các phần tử của chuỗi và đếm số lần xuất hiện của từng ký tự
  - **Cách tiếp cận tốt hơn**:
    - Sử dụng kỹ thuật băm bằng cách dùng một mảng có kích thước là 26 để chứa giá trị là số lần xuất hiện của một ký tự trong chuỗi
    - Dùng hàm băm để tính toán ra chỉ số của ký tự trong chuỗi
    - Duyệt qua toàn bộ chuỗi và tăng giá trị của phần tử mảng có chỉ số tương ứng bằng với chỉ số của ký tự vừa được tính ở bước trên

Bảng băm – Hash table

- Kỹ thuật dây chuyền (Separate chaining)
  - Kĩ thuật dây chuyền là một trong những kĩ thuật phổ biến nhất để giải quyết xung đột trong bảng băm
  - Sử dụng các danh sách liên kết → mỗi phần tử của bảng băm là một danh sách liên kết
  - Các phần tử được chèn vào các danh sách liên kết trong bảng băm. Khi có xung đột xảy ra, tức là các phần tử có cùng mã băm (hash code) thì được chèn cùng vào một danh sách liên kết

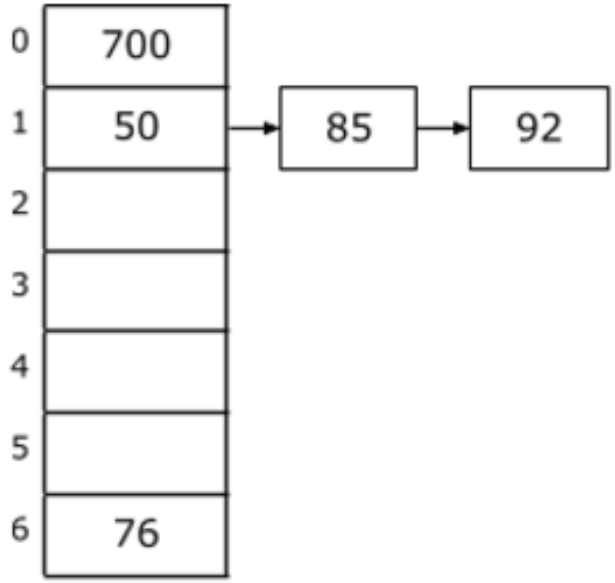
Bảng băm – Hash table

- Kỹ thuật dây chuyền (Separate chaining)
  - **Ví dụ**: có hàm băm chuyển đổi các khóa **50, 700, 76, 85, 92, 73, 101** bằng cách chia cho 7 rồi lấy số dư

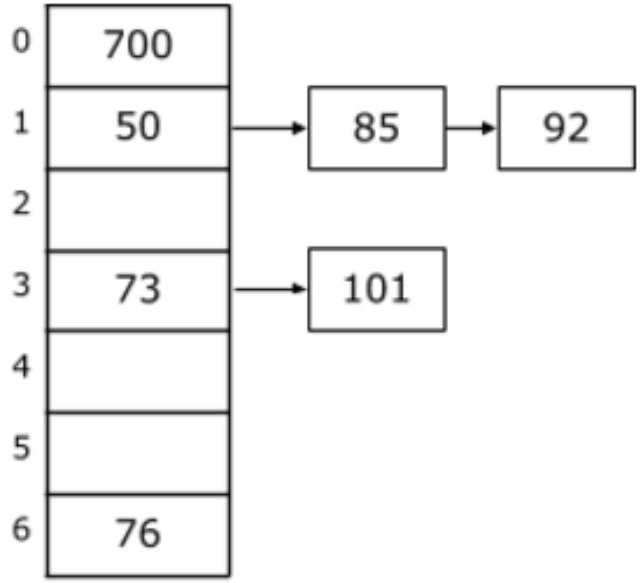


Bảng băm – Hash table

- Kỹ thuật dây chuyền (Separate chaining)
  - **Ví dụ**: có hàm băm chuyển đổi các khóa **50, 700, 76, 85, 92, 73, 101** bằng cách chia cho 7 rồi lấy số dư



Chèn 92, xung đột xảy ra, chèn phần tử mới vào chuỗi



Chèn 73 và 101

Bảng băm – Hash table

- Kỹ thuật dây chuyền (Separate chaining)
  - Để chèn một phần tử vào bảng băm, ta cần tìm chỉ số băm cho khóa đã cho. Giả sử chỉ số được tính toán theo công thức sau

    index = key % number_of_rows
  - **Thao tác chèn**: chuyển tới hàng tương ứng với chỉ số tính toán được theo công thức ở trên và chèn phần tử mới vào bảng như một nút mới ở cuối danh sách liên kết
  - **Thao tác xóa**: Để xóa một nút khỏi bảng băm, đầu tiên ta cần tính toán chỉ số cho khóa và chuyển đến hàng tương ứng với chỉ số đó, tìm kiếm phần tử với khóa đã cho trên danh sách liên kết và xóa nó nếu tìm thấy

Bảng băm – Hash table

- Kỹ thuật dây chuyền (Separate chaining)
  - Ưu điểm
    - Cài đặt đơn giản
    - Không phải quan tâm tới kích thước bảng băm và luôn thêm được dữ liệu vào bảng bằng cách thêm vào các danh sách liên kết
    - Phương pháp này thường được sử dụng khi ta không biết tần suất dữ liệu được thêm vào và xóa ra khỏi bảng

Bảng băm – Hash table

- Kỹ thuật dây chuyền (Separate chaining)
  - Nhược điểm
    - Hiệu năng của phương pháp này không tốt bằng phương pháp đánh địa chỉ mở vì với phương pháp đánh địa chỉ mở thì mọi dữ liệu đều được chứa trong cùng một bảng băm mà không cần trỏ tới một vùng nhớ ngoài bảng
    - Đôi khi lãng phí bộ nhớ (có nhiều ô trống do các khóa trùng nhau được đưa vào cùng một danh sách liên kết), đôi khi chúng không bao giờ được sử dụng tới
    - Khi mà chuỗi (danh sách liên kết) trở nên quá dài, lúc đó thời gian cho các thao tác tìm kiếm, xóa phần tử có thể rất tốn thời gian
    - Cần thêm bộ nhớ cho các phần tử của danh sách liên kết

Bảng băm – Hash table

- Hiệu năng của kỹ thuật dây chuyền
  - Giả sử mỗi khóa (key) được chèn đồng đều vào các ô nhớ chứa dữ liệu của bảng băm

m = Số lượng ô nhớ trong bảng băm

n = Số lượng khóa được chèn vào bảng

Hệ số tải $a = n/m$

Thời gian kỳ vọng cho thao tác tìm kiếm = $O(1+a)$

Thời gian kỳ vọng cho thao tác chèn và xóa dữ liệu = $O(1+a)$

Nếu $a = 1$ thì lúc đó ta coi thời gian cho các thao tác tìm kiếm, chèn và xóa dữ liệu là O(1)

Bảng băm – Hash table

- Kỹ thuật đánh địa chỉ mở - Open Addressing
  - Ý tưởng của băm địa chỉ mở (open addressing) là mỗi ô của bảng chỉ lưu duy nhất một phần tử, do đó ta không cần danh sách móc nối
  - Xung đột sẽ được băm địa chỉ mở giải quyết bằng cách sử dụng m hàm băm độc lập h_0, h_1, …, h_{m-1}, sao cho:

    Với bất kì phần tử x nào, m giá trị $h_0(x)$, $h_1(x)$, …, $h_{m-1}(x)$ đôi một khác nhau; do đó, $\{h_0(x), h_1(x), \ldots, h_{m-1}(x)\}$ là một hoán vị của $\{0, 1, \ldots, m-1\}$

  - Khi băm một khóa x, ta sẽ lần lượt kiểm tra từ ô $h_0(x)$ của bảng cho đến $h_{m-1}(x)$. Nếu tìm thấy một ô trống đầu tiên thì lưu x vào đó. Do $h_0(x)$, $h_1(x)$, …, $h_{m-1}(x)$ là một hoán vị của $\{0, 1, \ldots, m-1\}$, quá trình tìm kiếm ô trống luôn kết thúc sau tối đa m bước

Bảng băm – Hash table

- Kỹ thuật đánh địa chỉ mở - Open Addressing
  - Để tìm kiếm bảng băm địa chỉ mở ta sẽ thực hiện dò bảng (probing): bắt đầu từ vị trí $h_0(x)$ cho đến vị trí $h_{m-1}(x)$
    - Nếu x có trong bảng thì ta sẽ tìm được x ở một ô có địa chỉ $h_i(x)$ nào đó
    - Nếu x không chứa trong bảng, trong quá trình dò, ta sẽ bắt gặp một ô rỗng hoặc duyệt qua đến $h_{m-1}(x)$ mà vẫn chưa tìm được x

Bảng băm – Hash table

- Kỹ thuật đánh địa chỉ mở - Open Addressing
  - Trong thực tế, việc thiết kế *m* hàm băm ngẫu nhiên độc lập thỏa mãn mã băm đôi một khác nhau với một khóa cho trước là việc vô cùng khó. Cho dù ta có thực hiện được thì chi phí thời gian cũng không nhỏ
  - Trong thực tế, ta chấp nhận các hàm băm "phụ thuộc" với nhau ở một mức độ nào đó, mỗi mức độ cho chúng ta một phép dò khác nhau
  - Ta sẽ nghiên cứu: dò tuyến tính, dò nhị phân, dò bậc hai và băm kép

Bảng băm – Hash table

- Kỹ thuật dò tuyến tính – Linear probing
  - Trong phép dò tuyến tính, ta sẽ chỉ sử dụng một hàm băm tốt $h(x)$ để định nghĩa m hàm băm như sau:

$$h_i(x) = (h(x) + i) \bmod m,\ 0 \le i \le m-1$$

  - Khi thêm vào bảng băm, nếu chỉ mục đó đã có phần tử rồi thì giá trị chỉ mục sẽ được tính toán lại theo cơ chế tuần tự
  - VD:

```
index = index % hashTableSize
index = (index + 1) % hashTableSize
index = (index + 2) % hashTableSize
index = (index + 3) % hashTableSize
```

  - Điểm mạnh của phương pháp dò tuyến tính này là thực thi đơn giản. Tuy nhiên, các giá trị băm sẽ có xu hướng tụm lại với nhau thành một dãy con liên tục trên bảng băm. Ngoài ra, khi hệ số tải gần bằng 1 thì tìm kiếm với dò tuyến tính cực kì kém hiệu quả

Bảng băm – Hash table

- Kỹ thuật dò nhị phân – Binary probing
  - Dò nhị phân vừa lợi dụng điểm mạnh của dò tuyến tính, vừa có thể tính nhanh được trong thực tế
  - Chọn $m = 2^\ell$ cho phép chúng ta chuyển các thao tác nhân, chia, mod về các thao tác xử lí bít → có thể tính được hàm băm rất nhanh
  - Chọn $m = 2^\ell$ và sử dụng một hàm băm tốt $h(x)$ để định nghĩa m hàm băm:

$h_i(x) = (h(x) \oplus i)$, $0 \le i \le m-1$

trong đó $\oplus$ là phép XOR bít

Bảng băm – Hash table

- Kỹ thuật dò bậc hai – Quadratic probing
  - Dùng hàm bậc 2 để thiết kế *m* hàm băm

$$h_i(x) = (h(x) + i^2) \bmod m,\ 0 \le i \le m-1$$

  - Phương pháp dò bậc hai về mặt lý thuyết tốt hơn dò tuyến tính

Bảng băm – Hash table

- Băm kép – Double hashing
  - Băm kép sử dụng hai hàm băm độc lập $h(x)$, $g(x)$ để định nghĩa m hàm băm

$$h_i(x) = (h(x) + i*g(x)) \bmod m,\ 0 \le i \le m-1$$

  - Ví dụ:

```
index = (index + 1 * indexH) % hashTableSize;
index = (index + 2 * indexH) % hashTableSize;
```

  - Phương pháp này tốt hơn về mặt lý thuyết và trong thực tế sẽ chậm hơn

Bảng băm – Hash table

- Băm hoàn hảo
  - Mặc dù kì vọng thời gian tìm kiếm là O(1) (*giả sử hệ số tải là hằng số*), trong trường hợp xấu nhất, thời gian tìm kiếm có thể lên tới gần xấp xỉ $O(\log n)$ và đôi khi số này là một con số không hề nhỏ
  - Có thể làm giảm hiệu ứng xấu nhất đó bằng cách giảm hệ số tải → tăng kích thước của bảng băm, ví dụ $m = n^2$
  - Tuy nhiên, n^2 là một con số quá lớn và không thực tế. Chẳng hạn, chỉ băm 1000 phần tử mà cần tới 1 triệu bộ nhớ
  - Băm hoàn kết hợp cả nhận xét trên và ý tưởng của **kỹ thuật dây chuyền** để làm giảm thời gian tìm kiếm xấu nhất xuống O(1) mà bảng chỉ cần bộ nhớ n

Bảng băm – Hash table

- Băm hoàn hảo
  - Sử dụng hai hàm băm tốt {$h(x)$, $g(x)$} và bảng băm hai chiều T[1,2,…,m][…]
  - Mỗi hàng của bảng băm T[i] sẽ được coi như một bảng băm phụ, có kích thước phụ thuộc vào đầu vào
  - Khi băm vào bảng, ta thực hiện băm theo 2 pha:
    - *Pha đầu tiên*, sử dụng h để băm x vào hàng $h(x)$ của bảng T
    - *Pha thứ 2*, gọi C[i] là số lượng phần tử được băm cùng vào hàng thứ i sau pha đầu tiên. Với mỗi hàng i, ta cấp phát một bộ nhớ $C[i]^2$ cho hàng T[i] → coi hàng này như một bảng băm và dùng g để băm các phần tử x có cùng mã băm i vào ô $g(x)$ của hàng này. Đụng độ lần 2 (nếu có) sẽ được giải quyết sử dụng kỹ thuật dây chuyền

Bảng băm – Hash table

- Băm hoàn hảo
  - Một số lưu ý
    - Do bảng băm phụ có kích thước là bình phương số lượng phần tử được lưu trong hàng, đụng độ khi băm lần 2 này là O(1). Do đó, tìm kiếm có thể được thực hiện trong O(1)
    - Kích thước của các bảng băm con tương ứng với các hàng khác nhau có thể khác nhau
    - Bảng băm con thứ i (là T[i]) có kích thước $C[i]^2$. Do đó, khi băm vào bảng băm con T[i] trong pha 2 sử dụng hàm băm $g(x)$, địa chỉ thực sự trong bảng băm con là $g(x)$ mod $C[i]^2$
    - Người ta đã chứng minh rằng, khi hệ số tải $a = 1$ thì bộ nhớ của băm hoàn hảo là $2n$

Bảng băm – Hash table

- Thiết kế hàm băm trong thực tế
  - Ba phương pháp chính để giải quyết xung đột: kỹ thuật dây chuyền, địa chỉ mở và băm hoàn hảo → trong thực tế nên chọn phương pháp nào?
    - Trong các ứng dụng mà chúng ta phải thường xuyên thêm và xóa phần tử khỏi bảng, kỹ thuật dây truyền sẽ là một lựa chọn tốt
    - Trong các ứng dụng mà chúng ta chủ yếu thực hiện tìm kiếm, ít khi phải thêm hay xóa phần tử khỏi bảng (ví dụ ứng dụng từ điển chẳng hạn) thì băm hoàn hảo sẽ là một lựa chọn tốt
    - Nếu ứng dụng của chúng ta chủ yếu thực hiện tìm kiếm nhưng chúng ta lại có thêm thông tin về tần suất truy nhập khóa thì ta có thể sử dụng băm địa chỉ mở. Lưu ý là không nên để hệ số tải cao (lớn hơn 0.8)

Bảng băm – Hash table

- Ứng dụng
  - Được sử dụng để cài đặt một số cấu trúc dữ liệu trong C++ (unordered_set, unordered_map), Java, C#, Python
  - Với các bài toán đặc thù → sẽ phải tự viết hàm băm riêng và xây dựng cấu trúc dữ liệu bảng băm cho phù hợp
  - Bảng băm thường được ứng dụng trong
    - Lập chỉ mục Cơ sở dữ liệu
    - Tổ chức bộ nhớ đệm
    - Biểu diễn các đối tượng: Perl, Python, JavaScript và Ruby
    - Áp dụng trong các thuật toán để tăng tốc độ tính toán

Bảng băm – Hash table

- Bài tập
  1) Chèn dãy giá trị {4371, 1323, 6173, 4199, 4344, 9679, 1989} vào bảng băm với hàm băm hash(x) = x % 10:
     a) Bảng băm dây chuyền
     b) Bảng băm thăm dò tuyến tính
     c) Bảng băm thăm dò bậc hai
  2) Cài đặt cấu trúc unordered_set
  3) Cài đặt cấu trúc unordered_map
  4) Cài đặt từ điển dựa trên bảng băm thăm dò tuyến tính

Tìm kiếm trên dữ liệu phân nhánh

- Tìm kiếm
  - Một lớp lớn bài toán có thể phát biểu và giải quyết dưới dạng tìm kiếm
  - Yêu cầu tìm kiếm có thể là tìm những trạng thái, tính chất thỏa mãn một số điều kiện nào đó, hoặc tìm chuỗi hành động cho phép đạt tới trạng thái mong muốn
  - Ví dụ một số bài toán
    - Trò chơi
    - Lập lịch, hay thời khóa biểu
    - Tìm đường đi
    - Lập kế hoạch

Tìm kiếm trên dữ liệu phân nhánh

- Phát biểu bài toán tìm kiếm
  - Một vấn đề có thể giải quyết thông qua tìm kiếm bằng cách xác định tập hợp các phương án, đối tượng, hay trạng thái liên quan, gọi chung là không gian trạng thái (không gian tìm kiếm)
  - Thủ tục tìm kiếm sau đó sẽ khảo sát không gian trạng thái theo một cách nào đó để tìm ra lời giải cho vấn đề
  - Thuật toán tìm kiếm bắt đầu từ một trạng thái xuất phát nào đó → sử dụng những phép biến đổi trạng thái để nhận biết và chuyển sang trạng thái khác
  - Quá trình tìm kiếm kết thúc khi tìm ra lời giải, tức là khi đạt tới trạng thái đích

Tìm kiếm trên dữ liệu phân nhánh

- Phát biểu bài toán tìm kiếm
  - Bài toán tìm kiếm cơ bản có thể phát biểu thông qua năm thành phần chính
    - Tập các trạng thái Q → không gian trạng thái
    - Tập (không rỗng) các trạng thái xuất phát S ($S \subseteq Q$)
    - Tập (không rỗng) các trạng thái đích G ($G \subseteq Q$). Trạng thái đích có thể được cho một cách tường minh hoặc không tường minh (điều kiện mà trạng thái đích cần thỏa)
    - Các toán tử, còn gọi là hành động hay chuyển động hay hàm chuyển tiếp → cho phép chuyển từ trạng thái hiện thời sang các trạng thái khác
    - Giá thành hoặc chi phí c

    ***Lời giải* là chuỗi chuyển động cho phép di chuyển từ trạng thái xuất phát tới trạng thái đích hoặc chỉ là trạng thái đích tùy theo yêu cầu của bài toán**

Tìm kiếm trên dữ liệu phân nhánh

- Phân loại thuật toán tìm kiếm
  - **Tìm kiếm mù**: Với một trạng thái cho trước, để tìm ra trạng thái tiếp theo, hoàn toàn không có thông tin nên việc chọn ra trạng thái thường theo một thứ tự cho trước, nên phương pháp này chỉ phù hợp với những bài toán có ít trạng thái
  - **Tìm kiếm có thông tin**: So với tìm kiếm mù, thuật toán sẽ tìm kiếm những trạng thái tối ưu dựa trên những thông tin cho trước, nên việc tìm ra trạng thái tối ưu sẽ nhanh hơn
  - **Tìm kiếm cục bộ**: Đối với thuật toán này thường không cần thiết phải lưu lại các trạng thái, thích hợp hơn so với tìm kiếm có thông tin đối với các bài có số lượng trạng thái cực lớn trong thực tế → dùng để giải quyết các bài toán tối ưu khó

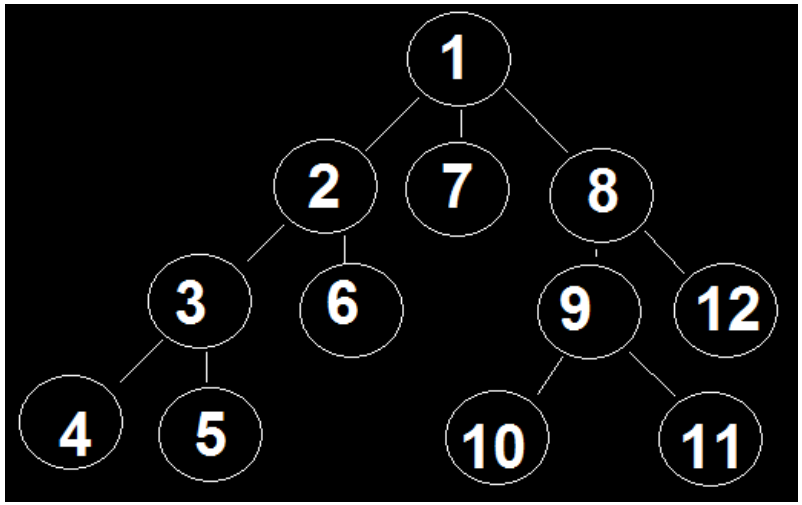
Tìm kiếm trên dữ liệu phân nhánh

- Tìm kiếm theo chiều rộng trên cây
  - Thuật toán tìm kiếm theo chiều rộng (Breadth First Search - BFS) là một thuật toán duyệt hoặc tìm kiếm một phần tử trên một cấu trúc dữ liệu dạng cây hay một đồ thị
  - BFS sẽ ưu tiên theo chiều ngang, nghĩa là duyệt từ trái qua phải hết rồi mới duyệt tiếp xuống dưới cho từng phần tử

Tìm kiếm trên dữ liệu phân nhánh

- Tìm kiếm theo chiều rộng trên cây
  - Ví dụ: thứ tự đi trong hình minh họa như sau

1. Bắt đầu từ 1 => 2 => 7 => 8 => hết nút ngang
2. Tiếp tục 2 => 3 => 6 hết nút ngang.
3. Bắt đầu 7 => không có nút ngang.
4. Tiếp tục 8 => 9 => 12 => hết nút ngang
5. Bắt đầu 3 => 4 => 5 => hết nút ngang
6. Tiếp tục 6 => không có nút ngang
7. Bắt đầu 9 => 10 => 11 hết nút ngang
8. Tiếp tục 12 => không có nút ngang
9. Bắt đầu 4 => hết nút
10. Tiếp tục 5 => hết nút
11. Bắt đầu 10 => hết nút
12. Tiếp tục 11 => hết nút



Tìm kiếm trên dữ liệu phân nhánh

- Thuật toán tìm kiếm theo chiều rộng trên cây

B1. Tạo một queue rỗng bfqueue

B2. Enqueue nút gốc vào bfqueue

B3. Vòng lặp while khi queue không rỗng

B3.1. Gán nút tạm current bằng phần tử đầu tiên của queue

B3.2. Dequeue phần tử đầu tiên từ bfqueue

B3.3. In ra dữ liệu của nút current

B3.4. Enqueue các con của current vào bfqueue từ trái qua phải

Tìm kiếm trên dữ liệu phân nhánh

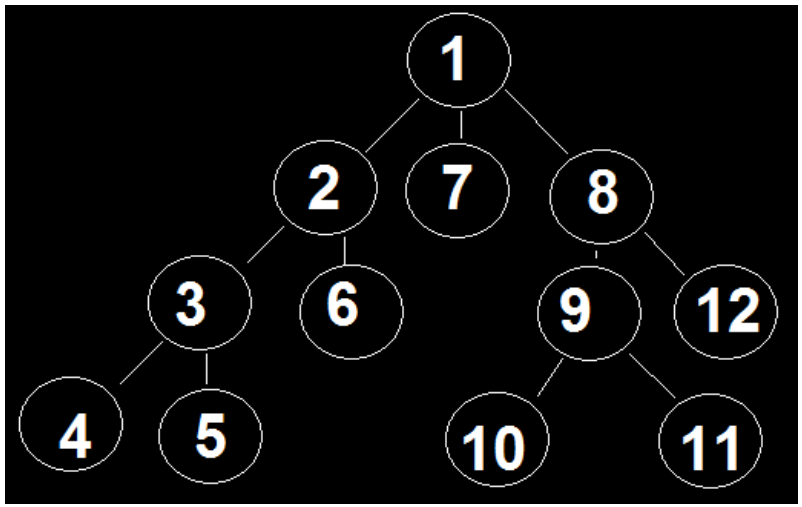
- Tìm kiếm theo chiều rộng trên cây
  - Các tính chất thuật toán BFS
    - Sử dụng cấu trúc dữ liệu hàng đợi để lưu trữ các nút
    - Là cấu trúc dữ liệu dạng dạng cây hay đồ thị
    - Độ phức tạp thời gian là: O(|V| + |E|) với V và E lần lượt là số đỉnh và số cạnh duyệt qua
    - Độ phức tạp không gian là O(|V|)

Tìm kiếm trên dữ liệu phân nhánh

- Tìm kiếm theo chiều sâu trên cây
  - Thuật toán tìm kiếm theo chiều sâu (Depth First Search - DFS) là một thuật toán duyệt hoặc tìm kiếm một phần tử trên một cấu trúc dữ liệu dạng cây hay một đồ thị
  - DFS bắt đầu đi từ một đỉnh của cây, sau đó từ đỉnh đó tìm ra nút đầu tiên và cứ tìm các nút tiếp theo cho đến khi không đi được nữa thì lại quay về nút trước đó và tìm sang nút bên cạnh để đi tiếp
  - DFS giống như đi trong mê cung, khi gặp ngõ cụt thì qua trở lại và thử đi theo một hướng khác

Tìm kiếm trên dữ liệu phân nhánh

- Tìm kiếm theo chiều sâu trên cây
  - Ví dụ: thứ tự đi trong hình minh họa như sau



1. Bắt đầu từ 1 => 2 => 3 => 4 => hết đường đi
2. Quay lại 3 => 5 => hết đường đi
3. Tiếp tục từ 3 quay lại 2 => 6 => hết đường đi
4. Quay lại 2 => quay lại 1 => 7 => hết đường đi
5. Tiếp tục từ 1 => 8 => 9=> 10 => hết đường đi
6. Quay lại 9 => 11 => hết đường đi
7. Tiếp tục 9=> quay lại 8 => 12 => hết đường đi
8. Quay lại 8 => quay lại 1 => hết đường => KẾT THÚC

Tìm kiếm trên dữ liệu phân nhánh

- Thuật toán tìm kiếm theo chiều sâu trên cây

```
Khởi tạo một ngăn xếp S rỗng để lưu các nút
Với mỗi đỉnh u, khởi tạo u.visited = false
Đưa nút gốc (nút được viếng thăm đầu tiên) vào S
while S khác rỗng
    Lấy ra phần tử u khỏi ngăn xếp S
    If u.visited = false then
        u.visited = true
        Với mỗi hàng xóm chưa được viếng thăm w của u
            Đưa w vào S
Kết thúc tiến trình khi tất cả các nút đã được viếng thăm
```

Tìm kiếm trên dữ liệu phân nhánh

- Tìm kiếm theo chiều sâu trên cây
  - Các tính chất thuật toán DFS
    - Là cấu trúc dữ liệu dạng đồ thị, hay dạng cây
    - Độ phức tạp thời gian là: O(|V| + |E|) với V và E lần lượt là số đỉnh và số cạnh duyệt qua
    - Độ phức tạp không gian là O(|V|)

Tìm kiếm trên dữ liệu phân nhánh

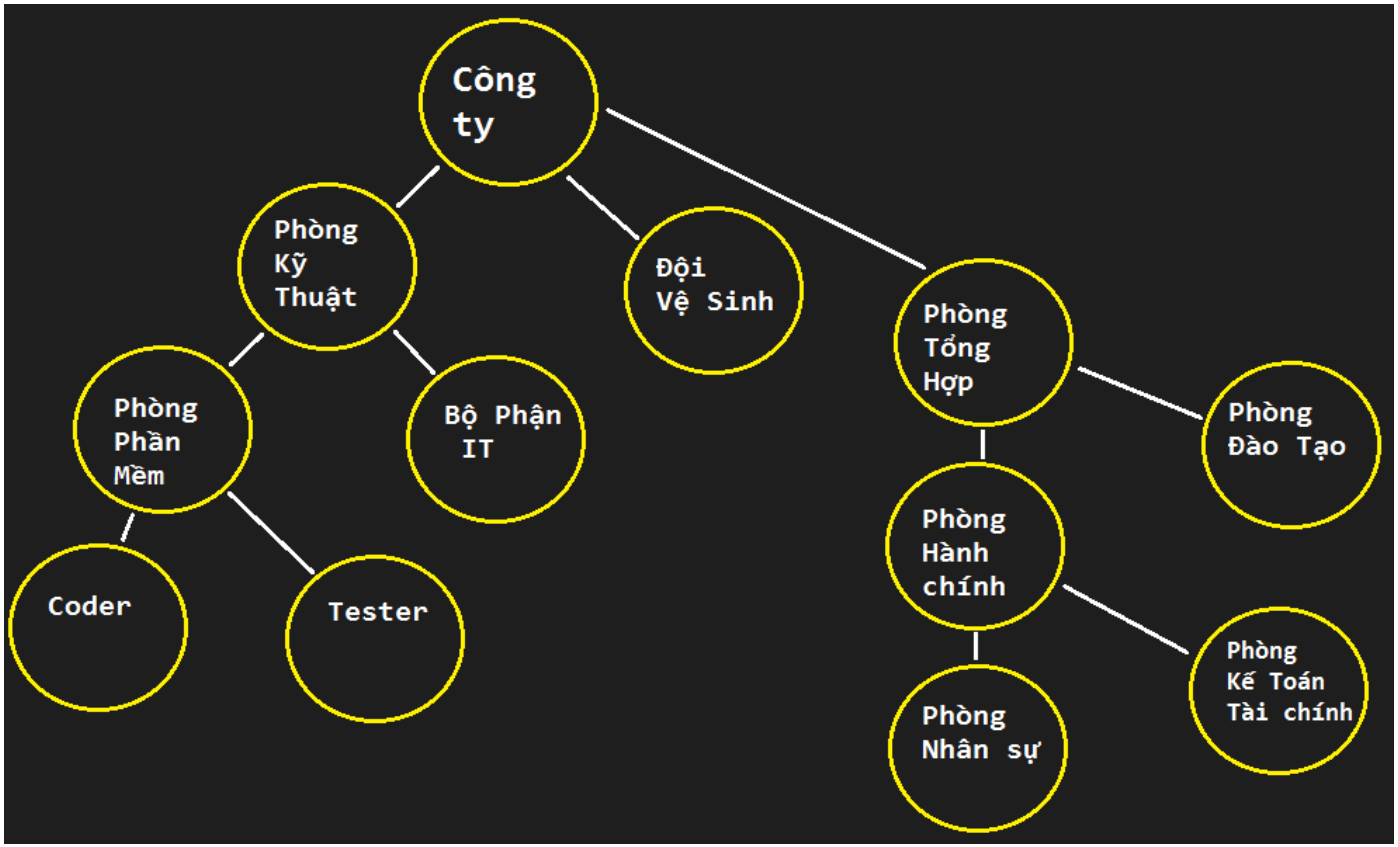
- Ưu và nhược điểm của BFS và DFS
  - Các thuật toán BFS và DFS là tìm kiếm mù, tức là tìm kiếm không theo sự hướng dẫn nào, phạm vi tìm kiếm được phát triển liên tục cho đến khi gặp trạng thái đích
  - BFS đảm bảo sẽ luôn tìm được nghiệm nếu bài toán có nghiệm, trong khi DFS không phải lúc nào cũng đảm bảo tìm thấy nghiệm
  - DFS càng ngày càng đi sâu hơn theo nhánh đang duyệt. Vậy chuyện gì sẽ xảy ra nếu nhánh đó là nhánh vô hạn?
  - Trong nhiều trường hợp DFS tìm được kết quả nhanh hơn BFS, không gian bộ nhớ cần cho việc lưu trữ cũng ít hơn
  - Đối với các chiến lược tìm kiếm mù đó là việc bùng nổ tổ hợp trong quá trình tìm kiếm. Càng về sau, số lượng trạng thái chờ để được xét duyệt trở nên rất lớn khiến cho không gian lưu trữ cũng lớn theo

Tìm kiếm trên dữ liệu phân nhánh

- Ứng dụng của các thuật toán tìm kiếm trên cây
  - Tìm kiếm trong không gian trạng thái
    - Các nút của cây chính là các trạng thái của không gian trạng thái
    - Nút gốc là trạng thái ban đầu. Quá trình tìm kiếm sẽ ngừng khi gặp tập các trạng thái kết thúc (các trạng thái đích)
    - Mỗi chiến lược tìm kiếm sẽ có cách duyệt cây tìm kiếm khác nhau

Tìm kiếm trên dữ liệu phân nhánh

- Bài tập
  - Áp dụng vào thực tế dự án
    - Cho sơ đồ tổ chức của một công ty như sau



Tìm kiếm trên dữ liệu phân nhánh

- Bài tập
  - Áp dụng vào thực tế dự án
    - Xây dựng mô hình quản lý nhân viên một công ty phần mềm theo cấu trúc dạng cây như sơ đồ
    - Chương trình phải đáp ứng các yêu cầu như sau

      => Duyệt được toàn bộ thông tin các phòng ban và in ra văn bản

      => Nhập tên một nhân viên bất kỳ hoặc mã id hoặc một thông tin nào đó thì phải tìm được nhân viên đó thuộc phòng ban nào

Tìm kiếm trên dữ liệu phân nhánh

- Bài tập
  - Cài đặt thuật toán tô màu loang (Flood fill)

    Một ảnh được biểu diễn bởi một mảng hai chiều các số nguyên, mỗi số nguyên biểu diễn giá trị điểm ảnh của ảnh đó (từ 0 đến 65535).

    Cho một tọa độ (sr, sc) biểu diễn điểm ảnh xuất phát (hàng và cột) của thuật toán flood fill, và một giá trị màu mới newColor dùng để loang màu trong ảnh.

    Để thực hiện một vết loang, xem xét điểm ảnh xuất phát, thêm mọi điểm ảnh theo 4 hướng có cùng màu và kề với điểm ảnh xuất phát, tiếp theo thêm mọi điểm ảnh kề với các điểm ảnh đó (cũng cùng màu với điểm ảnh xuất phát), và cứ như vậy. Thay thế màu của tất cả các điểm ảnh được đề cập bê trên bằng màu mới newColor.

    Cuối cùng, trả lại ảnh đã được sửa đổi.

Tìm kiếm tối ưu, nhánh và cận

- Thuật toán Best First Search
  - Thuộc loại thuật toán tìm kiếm kinh nghiệm
  - Dùng hàm đánh giá để hướng dẫn tìm kiếm
  - Là tìm kiếm theo chiều rộng (Breadth First Search) được hướng dẫn bởi hàm đánh giá
  - Tư tưởng của thuật toán này là việc tìm kiếm bắt đầu tại nút gốc và tiếp tục bằng cách duyệt các nút tiếp theo có giá trị của hàm đánh giá là thấp nhất so với các nút còn lại nằm trong hàng đợi

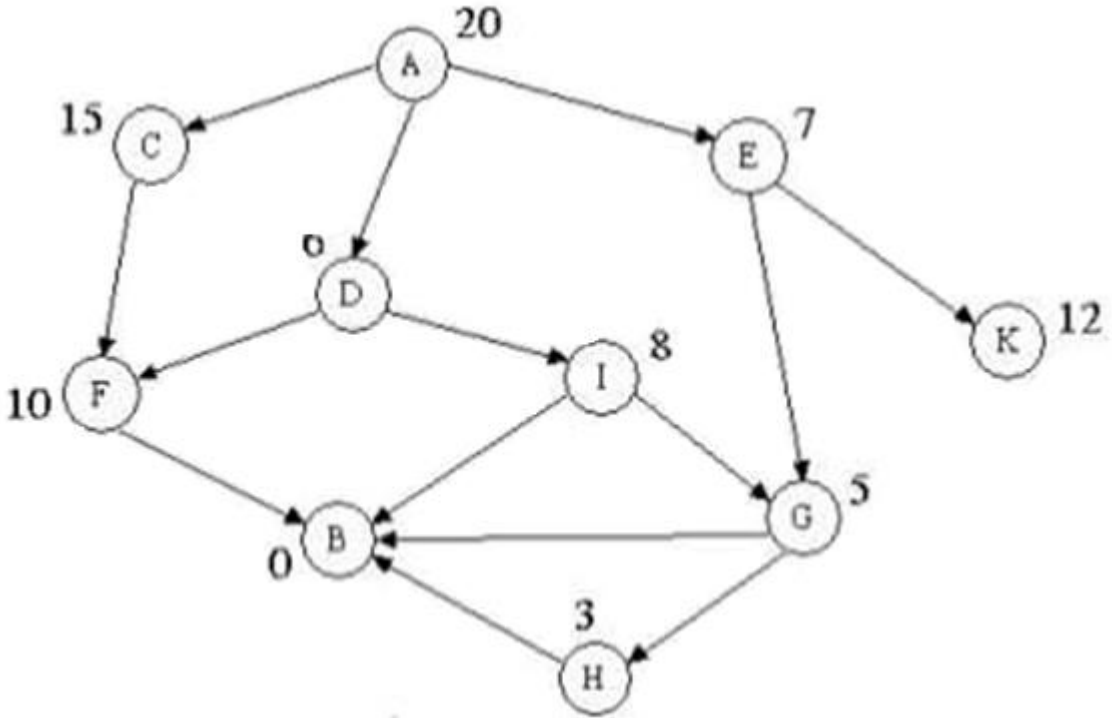
Tìm kiếm tối ưu, nhánh và cận

- Thuật toán Best First Search
  - Mã giả

```
void Best_First_Search() {
  Khởi tạo danh sách L chỉ chứa trạng thái ban đầu;
  loop do
    if L rỗng then
      {thông báo thất bại; stop};
    Loại trạng thái u ở đầu danh sách L;
    if u là trạng thái kết thúc then
      {thông báo thành công; stop};
    for mỗi trạng thái v kề u do
      Xen v vào danh sách L sao cho L được sắp theo thứ tự tăng dần
      của hàm đánh giá;
}
```

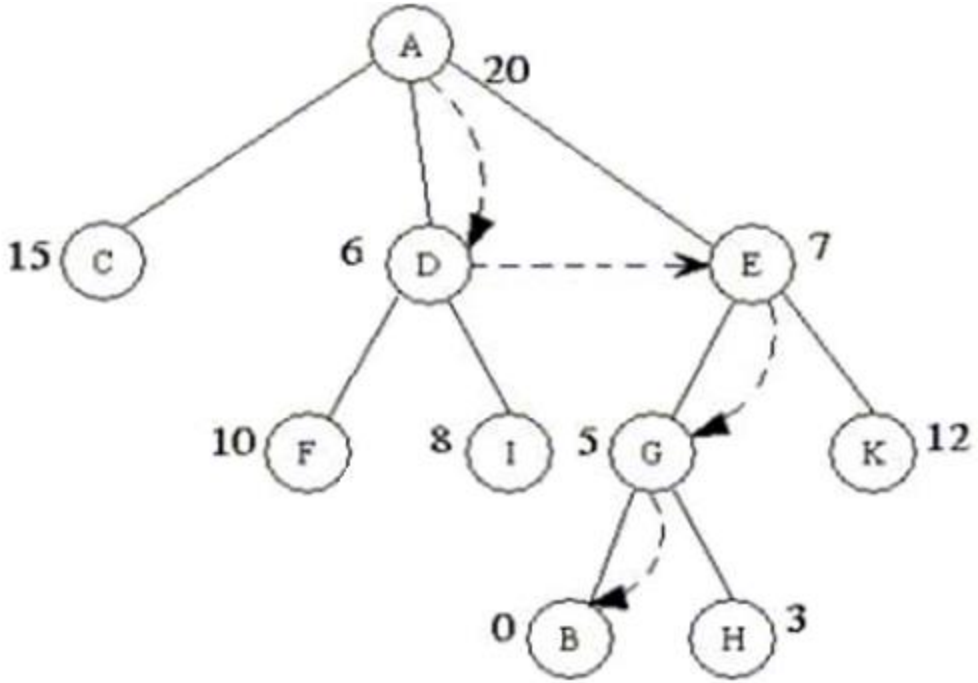
Tìm kiếm tối ưu, nhánh và cận

- Thuật toán Best First Search
  - Ví dụ: Cho đồ thị như hình dưới với nút gốc là A, duyệt và tìm đường đi tốt nhất đến B



Tìm kiếm tối ưu, nhánh và cận

- Thuật toán Best First Search
  - Đầu tiên, phát triển đỉnh A với các đỉnh kề là C, D và E
  - Đỉnh D có giá trị hàm đánh giá nhỏ nhất nên nó được chọn để phát triển và sinh ra F, I. Trong số các đỉnh chưa được phát triển C, E, F, I thì đỉnh E có giá trị đánh giá nhỏ nhất, nó được chọn để phát triển và sinh ra các đỉnh G, K. Vì G tốt nhất → phát triển G sinh ra B, H → gặp B và kết thúc



Tìm kiếm tối ưu, nhánh và cận

- Phương pháp nhánh - cận
  - Chuyển đến slide về bài toán nhánh – cận

Tìm kiếm tối ưu theo kinh nghiệm

- Các thuật toán tìm kiếm theo kinh nghiệm
  - Thuật toán leo đồi (hill climbing)
  - Thuật toán mô phỏng tôi luyện (Simulated annealing)
  - Thuật toán di truyền (Genetic algorithm)
  - Thuật toán tối ưu bầy đàn (Particle swarm optimization)
  - Thuật toán tối ưu đàn kiến (Ant colony optimization)
  - Thuật toán tối ưu bầy ong (Artificial bee colony)
  - Thuật toán tối ưu ruồi dấm (Fruit-fly optimization)

→ Các thuật toán tính toán mềm (Soft computing)

Tìm kiếm tối ưu theo kinh nghiệm

- Các thuật toán tìm kiếm theo kinh nghiệm
  - **Meta-heuristic**
    - **Phương pháp Heuristic** để giải quyết một lớp các vấn đề tính toán rất chung bằng cách kết hợp các kinh nghiệm do người dùng cung cấp với hi vọng đưa ra được một quy trình hiệu quả hơn
    - **Thuật toán metaheuristic** sử dụng nhiều **heuristic** kết hợp với các **kỹ thuật phụ trợ** nhằm khai phá không gian tìm kiếm
    - Mỗi **thuật toán metaheuristic** tổng quát là một lược đồ tính toán đề xuất cho lớp bài toán rộng, khi dùng cho các bài toán cụ thể cần thêm các vận dụng chi tiết cho phù hợp
    - Kỹ thuật tính toán mềm để giải quyết các vấn đề tối ưu hóa rời rạc

Tìm kiếm tối ưu theo kinh nghiệm

- Các thuật toán tìm kiếm theo kinh nghiệm
  - Thuật toán đa thức và thuật toán hàm mũ
    - **Thuật toán đa thức** là thuật toán có độ phức tạp về thời gian trong trường hợp xấu nhất của nó là **đa thức**. Nó còn được gọi là thuật toán "nhanh"
    - **Thuật toán hàm mũ** là thuật toán có độ phức tạp về thời gian trong trường hợp xấu nhất của nó là **hàm mũ**. Nó còn được gọi là thuật toán "chậm"

Tìm kiếm tối ưu theo kinh nghiệm

- Các thuật toán tìm kiếm theo kinh nghiệm
  - Đánh giá thuật toán theo hai tiêu chí
  1. **Thuật toán giải quyết được yêu cầu của bài toán ở mức nào?**
     - Ví dụ, Thuật toán tìm được nghiệm **tối ưu** (nghiệm đúng) của bài toán hay chỉ là **nghiệm gần đúng** của bài toán
     - Nếu tìm được **nghiệm gần đúng** của bài toán, thì "gần" đến mức nào? Tức là phải chỉ ra được sự chênh lệch giữa **nghiệm đúng** và **nghiệm gần đúng**.
  2. **Độ phức tạp của thuật toán là thấp so với các thuật toán cùng đạt tiêu chí 1**
     - Một thuật toán để giải bài toán không phải khi nào cũng **đồng thời đạt được cả hai tiêu chí tốt**. Vì vậy tuỳ theo yêu cầu thực tế mà **xếp thứ tự ưu tiên** hai tiêu chí trên

Tìm kiếm tối ưu theo kinh nghiệm

- Các thuật toán tìm kiếm theo kinh nghiệm
  - **Lớp bài toán P, NP**

1. Với một bài toán, có hai khả năng xảy ra: Đã có lời giải, Chưa có lời giải
2. Với bài toán đã có lời giải, cũng có hai trường hợp xảy ra

   + Bài toán giải được bằng thuật toán

   + Bài toán không giải được bằng thuật toán
3. Với bài toán giải được bởi thuật toán cũng chia thành hai loại

   + Bài toán thực tế giải được ("**Dễ giải**"). Giải trong thời gian đủ nhanh, thực tế cho phép

   + Bài toán thực tế khó giải ("**Khó giải**"). Giải trong nhiều thời gian, thực tế khó chấp nhận

Tìm kiếm tối ưu theo kinh nghiệm

- Các thuật toán tìm kiếm theo kinh nghiệm
  - **Lớp bài toán P, NP**
  - ***Ký hiệu***:
  - Các bài toán lớp P: là lớp bài toán giải được bằng thuật toán đơn định (đơn trị), đa thức (Polynomial)

    → biết chắc rằng đã có thuật toán đơn định, đa thức để giải nó với nghiệm chính xác
  - Các bài toán lớp NP: là lớp bài toán giải được bằng thuật toán không đơn định (đa trị), đa thức
  - **Chú ý**: + P ⊂ NP, nhưng hiện nay người ta chưa biết P ≠ NP?

    + NP: Nondeterministic Polynomial

Tìm kiếm tối ưu theo kinh nghiệm

- Các thuật toán tìm kiếm theo kinh nghiệm
  - **Lớp bài toán NP- Hard, NP- Complete**
    - Các bài toán lớp NP-Hard (NP-khó): Bài toán A được gọi là NP-Hard nếu $\forall L \in NP$ đều là L "**quy dẫn**" về A

      (A *khó hơn* mọi bài toán L trong NP hay L là trường hợp riêng của A)

      *Một bài toán A được gọi là NP-khó (NP-hard) nếu như sự **tồn tại thuật toán đa thức** để giải nó kéo theo sự tồn tại thuật toán đa thức để giải mọi bài toán trong NP*

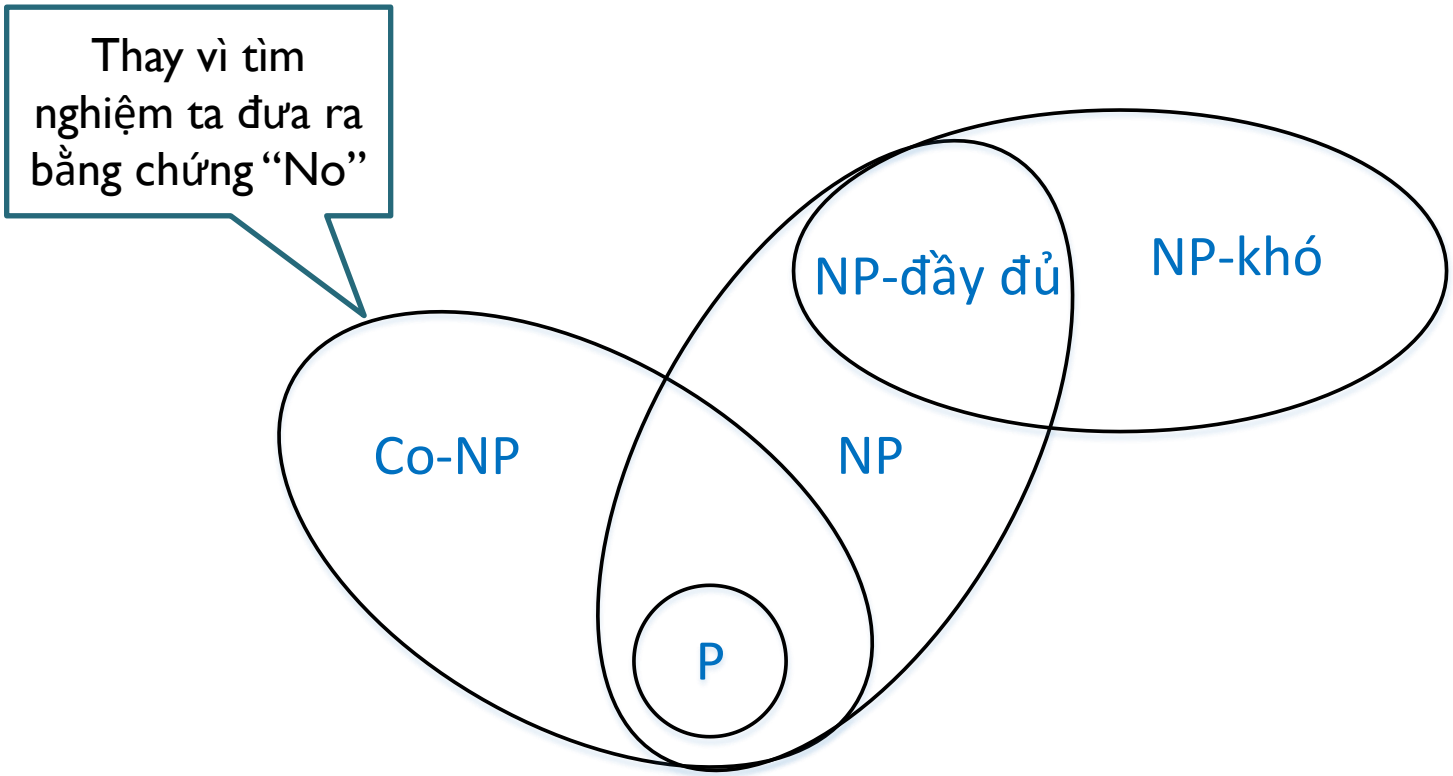
      → Các bài toán chỉ giải được bằng thuật toán đơn định với thời gian hàm mũ (chưa có lời giải trong thời gian đa thức)

Tìm kiếm tối ưu theo kinh nghiệm

- Các thuật toán tìm kiếm theo kinh nghiệm
  - **Lớp bài toán NP- Hard, NP- Complete**
    - Các bài toán lớp NP-Complete (NP-đầy đủ): nếu A là NP-Hard và $A \in NP$ → là các bài toán NP-Hard nằm trong lớp NP
    - Mỗi bài toán NP-đầy đủ đều là NP-khó. Tuy nhiên một bài toán NP-khó không nhất thiết phải là NP-đầy đủ
    - Strong NP-Hard: giới hạn dữ liệu đầu vào của bài toán NP-Hard để dễ giải hơn mà bài toán đã được giới hạn dữ liệu đầu vào đó vẫn là bài toán NP-Hard (bài toán tìm nghiệm gần đúng vẫn là bài toán NP-Hard)

Tìm kiếm tối ưu theo kinh nghiệm

- Các thuật toán tìm kiếm theo kinh nghiệm
  - **Lớp bài toán NP- Hard, NP- Complete**



Tìm kiếm tối ưu theo kinh nghiệm

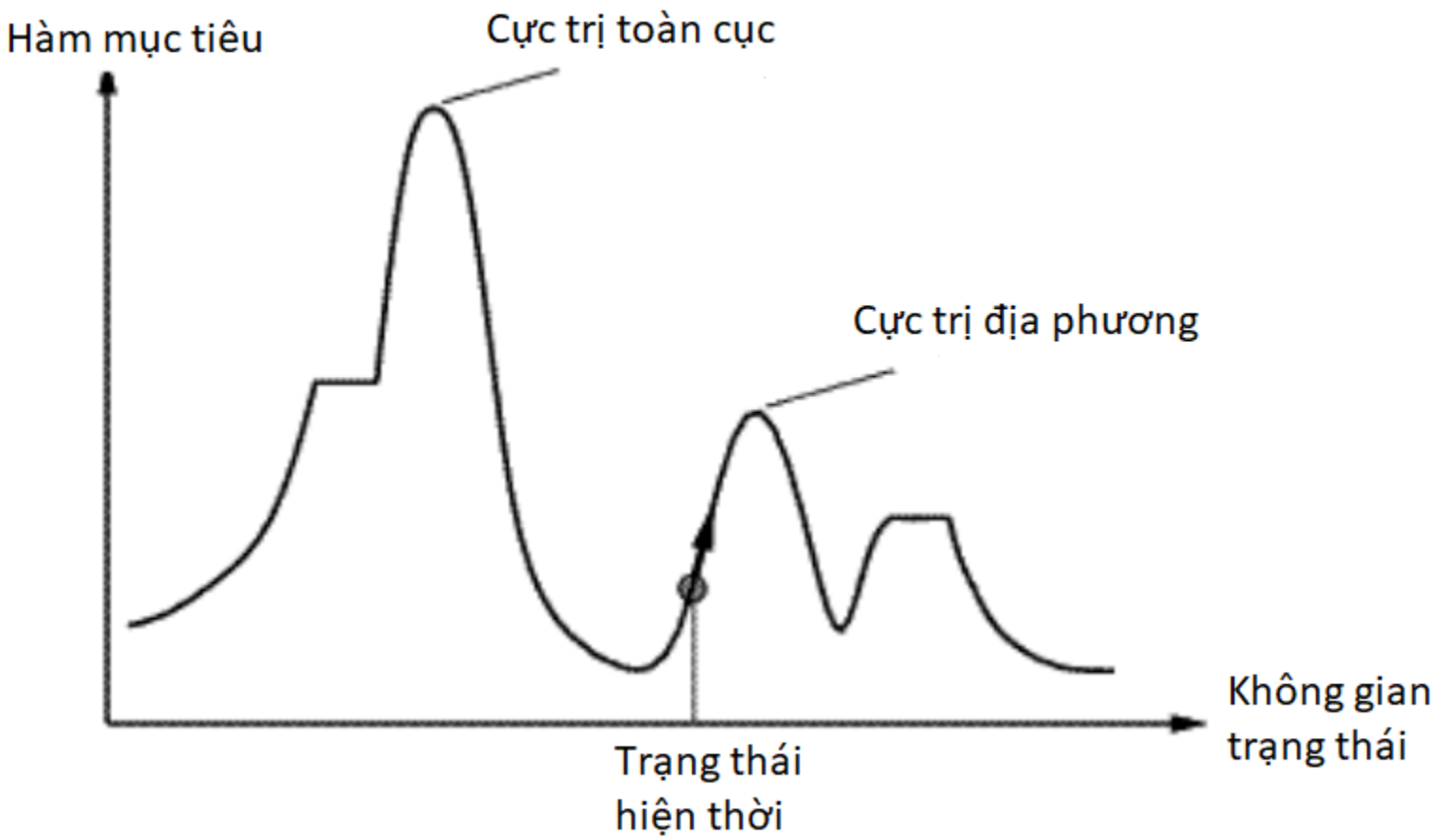
- Các thuật toán tìm kiếm theo kinh nghiệm
  - Có rất nhiều bài toán ứng dụng quan trọng thuộc vào lớp NP-khó
    - Vì khó hy vọng xây dựng được thuật toán đúng hiệu quả để giải chúng.
    - Một trong những hướng phát triển thuật toán giải các bài toán như vậy là xây dựng các thuật toán gần đúng
  - **Mục đích của các thuật toán tìm kiếm tối ưu theo kinh nghiệm**: giải các bài toán NP-khó và NP-đầy đủ → Tìm nghiệm **gần đúng** của Bài toán trong thời gian chấp nhận được
  - Phương án này phù hợp với thực tế: Xấp xỉ nhưng Nhanh!

Tìm kiếm tối ưu theo kinh nghiệm

- Thuật toán leo đồi
  - là một kỹ thuật tối ưu toán học thuộc họ tìm kiếm cục bộ
  - Thuật toán thực hiện bằng cách tạo ra hàng xóm cho trạng thái hiện thời và di chuyển sang hàng xóm có **hàm mục tiêu tốt hơn** → di chuyển lên cao đối với trường hợp cần cực đại hóa hàm mục tiêu
  - Thuật toán dừng lại khi đạt tới một đỉnh của đồ thị hàm mục tiêu, tương ứng với trạng thái không có hàng xóm nào tốt hơn
    - Có thể là đỉnh cao nhất → nghiệm tối ưu toàn cục
    - Có thể là những đỉnh thấp hơn → nghiệm tối ưu cục bộ

Tìm kiếm tối ưu theo kinh nghiệm

- Thuật toán leo đồi



Tìm kiếm tối ưu theo kinh nghiệm

- Thuật toán leo đồi
  - Di chuyển sang hàng xóm tốt nhất

Thuật toán

Đầu vào: bài toán tối ưu tổ hợp

Đầu ra: trạng thái với hàm mục tiêu lớn nhất tìm được

1. Chọn ngẫu nhiên trạng thái x
2. Gọi Y là tập các trạng thái hàng xóm của x
3. Nếu $\forall y_i \in Y$: Objective(y_i) < Objective(x) thì

   Kết thúc và trả lại x là kết quả
4. $x \leftarrow y_i$, trong đó $i = \text{argmax}_i$ (Objective(y_i))
5. Nhảy đến bước 2

Tìm kiếm tối ưu theo kinh nghiệm

- Thuật toán leo đồi
  - Leo đồi ngẫu nhiên

Thuật toán

Đầu vào: bài toán tối ưu tổ hợp

Đầu ra: trạng thái với hàm mục tiêu lớn nhất tìm được

1. Chọn ngẫu nhiên trạng thái x
2. Gọi Y là tập các trạng thái hàng xóm của x
3. Chọn ngẫu nhiên $y_i \in Y$
4. Nếu Objective(y_i) > Objective(x) thì

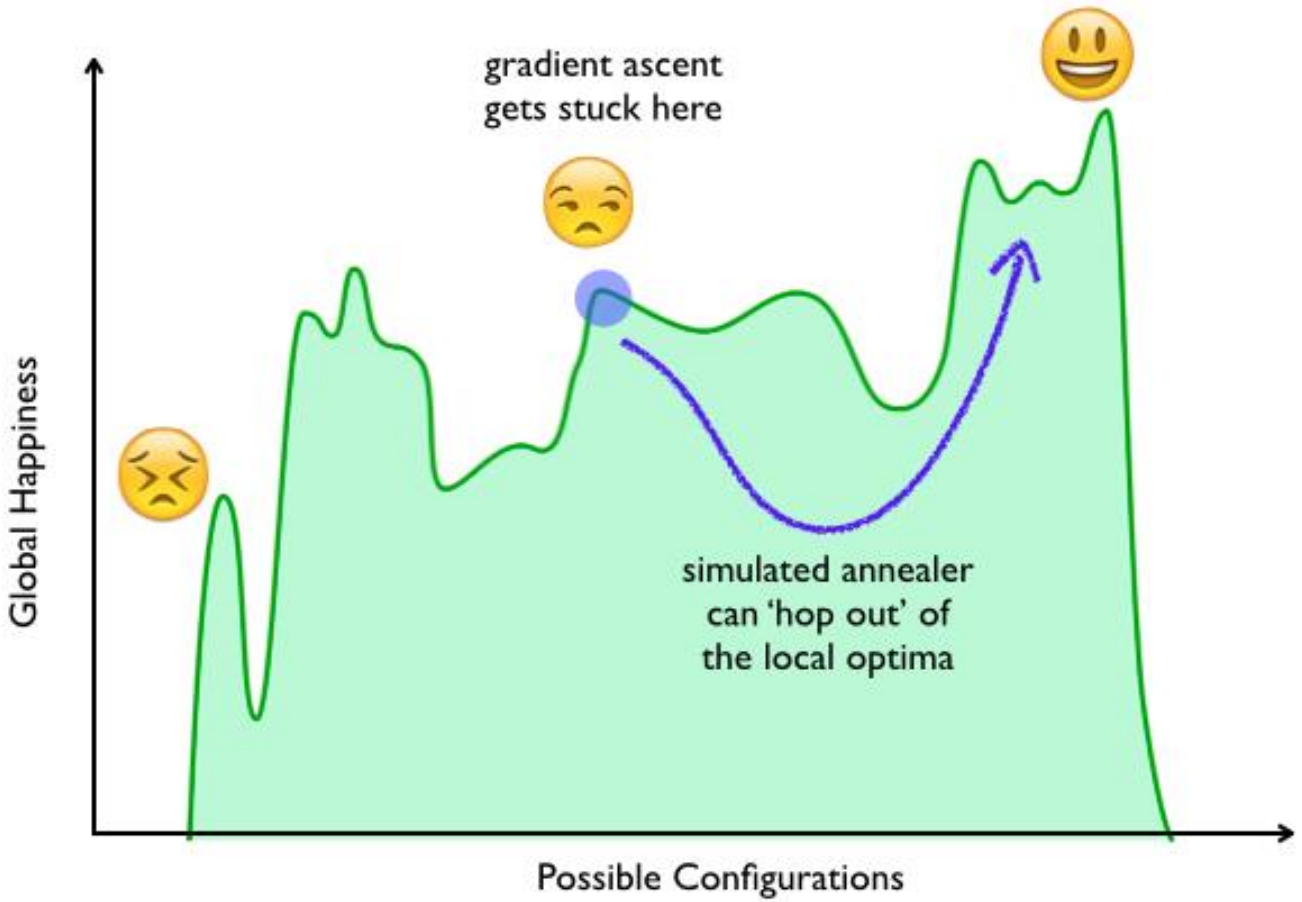
   $x \leftarrow y_i$
5. Nhảy đến bước 2

Tìm kiếm tối ưu theo kinh nghiệm

- Đặc điểm của thuật toán leo đồi
  - Đơn giản, dễ lập trình
  - Sử dụng ít bộ nhớ do không phải lưu lại các trạng thái. Tại mỗi thời điểm, thuật toán chỉ cần lưu lại trạng thái hiện thời và một trạng thái láng giềng
  - Dễ bị rơi vào lời giải tối ưu cục bộ → thực hiện thuật toán nhiều lần, mỗi lần sử dụng một trạng thái xuất phát sinh ngẫu nhiên khác nhau
  - Đối với những không gian có ít cực trị địa phương, leo đồi thường tìm được lời giải khá nhanh. Trong trường hợp không gian trạng thái phức tạp, thuật toán thường chỉ tìm được cực trị địa phương

Tìm kiếm tối ưu theo kinh nghiệm

- Thuật toán mô phỏng tôi luyện
  - Là một kỹ thuật leo đồi xác suất



Tìm kiếm tối ưu theo kinh nghiệm

- Thuật toán mô phỏng tôi luyện
  - Dựa trên nguyên lý làm nguội kim loại trong quá trình tôi luyện
  - Quá trình làm nguội bắt đầu từ nhiệt độ T_{max}, lúc này kim loại đang ở trạng thái lỏng
  - Sau khi ngừng đốt nóng, kim loại bắt đầu nguội dần và đạt tới nhiệt độ của môi trường xung quanh T_{min}, lúc này kim loại ở trạng thái rắn

Tìm kiếm tối ưu theo kinh nghiệm

- Thuật toán mô phỏng tôi luyện
  - Sau đây là các bước của thuật toán với **bài toán tối thiểu hóa năng lượng** E
  - **Bước 1**: Khởi tạo một trạng thái với năng lượng E_j, tỷ suất làm lạnh $\alpha \in [0, 1]$, nhiệt độ ban đầu $T = T_{max}$ với T_{max} không quá thấp để tránh rơi vào điểm tối ưu cục bộ và không quá cao để việc tìm kiếm không quá lâu
  - **Bước 2**: Tính toán sự thay đổi năng lượng giữa trạng thái hiện tại i và trước đó j của cấu hình

$$\Delta E = E_i - E_j$$

  - **Bước 3**: Nếu $\Delta E < 0$ thì trạng thái mới E_i được chấp nhận

Tìm kiếm tối ưu theo kinh nghiệm

- Thuật toán mô phỏng tôi luyện
  - **Bài toán tối thiểu hóa năng lượng** E
  - **Bước 3**: Nếu $\Delta E < 0$ thì trạng thái mới E_i được chấp nhận (up-hill).

    Ngược lai, tức $\Delta E > 0$ thì trạng thái mới E_i được chập nhận (down-hill) với xác suất $P = e^{-(\frac{\Delta E}{k_B T})}$, trong đó k_B là hằng số Boltzman

    → luật Metropolis acceptance rule
  - **Bước 4**: Nếu đạt điều kiện dừng thì kết thúc. Ngược lại, giảm nhiệt độ $T = \alpha T$ và trở lại bước 2

Tìm kiếm tối ưu theo kinh nghiệm

- Thuật toán mô phỏng tôi luyện
  - **Nhận xét:**
  - Khi nhiệt độ T càng nhỏ thì xác suất P sẽ càng nhỏ và tiến tới 0
  - Tức là khi thuật toán bắt đầu, ta sẵn sàng lựa chọn các phương án ít tối ưu hơn (chịu khó xuống dốc)
  - Khi càng về cuối vòng lặp, ta sẽ hạn chế dần việc chọn các phương án tồi (hạn chế xuống dốc) vì khi đó các trạng thái tốt đã dần được tìm ra

Tìm kiếm tối ưu theo kinh nghiệm

- Thuật toán mô phỏng tôi luyện
  - Mỗi phương án (nghiệm) tương ứng với một trạng thái của hệ thống
  - Hàm mục tiêu tương ứng với năng lượng mức năng lượng E
  - Hàng xóm (Neighborhood) của E_i là chuỗi các trạng thái tiếp sau mà trạng thái hiện tại có thể đạt tới
  - Tham số điều khiển chính là nhiệt độ T
  - Phương án tối ưu chấp nhận được (tối ưu địa phương) tại nhiệt độ nền (bằng với nhiệt độ của môi trường xung quanh)

Tìm kiếm tối ưu theo kinh nghiệm

- Thuật toán di truyền
  - Là thuật toán tìm kiếm được thiết kế dựa trên sự tương tự với quá trình chọn lọc tự nhiên và thuyết tiến hoá của Charles Darwin
  - Cho lời giải tốt trong nhiều bài toán tối ưu và được sử dụng rộng rãi trong rất nhiều ứng dụng khác nhau
  - **Ý tưởng**: học tập từ quá trình **sinh tồn** và **chọn lọc tự nhiên** của sinh vật trong tự nhiên, trong đó cá thể với **khả năng thích nghi cao hơn** sẽ chiếm ưu thế và tồn tại. Nếu ta coi đây là bài toán tối ưu thì quá trình tiến hoá sẽ sinh ra **cá thể tối ưu**, tức là **cá thể thích nghi cao**

Tìm kiếm tối ưu theo kinh nghiệm

- Thuật toán tối ưu bầy đàn
  - Hãy tưởng tượng một đàn chim bay lượn quanh một khu vực mà chúng có thể ngửi thấy một nguồn thức ăn tiềm ẩn
  - Con nào gần thức ăn nhất kêu to nhất và những con khác xoay vòng về hướng của nó
  - Nếu bất kỳ con chim bay vòng nào khác đến gần mục tiêu hơn con đầu tiên, nó kêu to hơn và những con khác quay về phía nó
  - Mô hình siết chặt này tiếp tục cho đến khi một trong những con chim xuất hiện tại nguồn thức ăn

Tìm kiếm tối ưu theo kinh nghiệm

- Thuật toán tối ưu bầy đàn
  - Thuật toán tối ưu bầy đàn (Particle swarm optimization - PSO) được đề xuất bởi Kennedy và Eberhart năm 1995
  - Ý tưởng chính của thuật toán này dựa trên cách thức các loài chim di chuyển để cố gắng tìm nguồn kiếm thức ăn hoặc tương tự hành vi của một đàn cá

Các con chim không có tri thức nào về nguồn thức ăn nhưng chúng biết **nguồn thức ăn cách vị trí hiện tại bao xa**

Vậy chiến lược nào tốt nhất để tìm thức ăn?

→ Theo con chim gần với thức ăn nhất

Tìm kiếm tối ưu theo kinh nghiệm

- Thuật toán tối ưu bầy đàn
  - Ban đầu, Russell Eberhart and James Kennedy bắt đầu phát triển phần mềm máy tính mô phỏng các đàn chim bay quanh các nguồn thức ăn, sau đó sau đó nhận ra rằng các thuật toán của họ hoạt động tốt như thế nào đối với các bài toán tối ưu hóa
  - Qua một số lần lặp, một nhóm các biến được điều chỉnh giá trị của chúng gần hơn với biến có giá trị gần với mục tiêu nhất tại bất kỳ thời điểm nhất định nào
  - Là một thuật toán đơn giản và dễ cài đặt

Tìm kiếm tối ưu theo kinh nghiệm

- Thuật toán tối ưu bầy đàn
  - Sự di chuyển của mỗi particle bị ảnh hưởng bởi vị trí tốt nhất của nó cho đến hiện tại, nhưng được dẫn hướng tới các vị trí được biết đến tốt nhất trong không gian tìm kiếm, được cập nhật thành vị trí tốt hơn được tìm thấy bởi các particle khác. Điều này được kỳ vọng sẽ di chuyển bầy đàn đến những phương án tốt nhất
  - PSO là metaheuristic vì nó đưa ra ít hoặc không có giả định nào về bài toán đang được tối ưu và có thể tìm kiếm trong không gian rất lớn các phương án ứng viên
  - Tuy nhiên, metaheuristics như PSO không đảm bảo tìm được phương án tối ưu

Tìm kiếm tối ưu theo kinh nghiệm

- Thuật toán theo dõi ba biến toàn cục:
  - Giá trị hoặc điều kiện mục tiêu
  - Giá trị tốt nhất toàn cục (gBest) cho biết dữ liệu của particle nào hiện gần với Mục tiêu nhất → tiến dần dần tới mục tiêu
  - Giá trị dừng cho biết khi nào thuật toán sẽ dừng nếu không tìm thấy Mục tiêu
- Mỗi particle bao gồm
  - Dữ liệu đại diện cho một phương án khả thi (vị trí của particle)
  - Giá trị Vận tốc cho biết mức độ Dữ liệu (vị trí) có thể được thay đổi
  - Giá trị cá nhân tốt nhất (pBest) cho biết Dữ liệu của particle gần nhất với Mục tiêu (giá trị tốt nhất mà particle tìm thấy cho tới hiện tại)

Tìm kiếm tối ưu theo kinh nghiệm

- Thuật toán tối ưu bầy đàn
  - **Cá thể** được gọi là **particle** và **dân số** được gọi là **swarm**. Mỗi cá thể chứa một phương án tiềm năng

    → Duy trì nhiều phương án tiềm năng tại một thời điểm
  - Mỗi **particle** x_i trong **swarm** di chuyển trong không gian tìm kiếm với một **tốc độ** (velocity) được tính toán bởi **phương án tốt nhất của nó** và **phương án tốt nhất trong swarm**
  - Sau mỗi lần di chuyển, mỗi particle x_i được đánh giá bằng một hàm mục tiêu f(x_i) để xác định tính thích nghi của nó

Tìm kiếm tối ưu theo kinh nghiệm

- Thuật toán tối ưu bầy đàn
  - Giả sử có một swarm $S = \{x_1, x_2, \ldots, x_N\}$
  - X_i^t là vị trí của particle *i* trong không gian tìm kiếm ở thế hệ thứ *t* và được cập nhật bởi:

$$X_i^{t+1} = X_i^t + V_i^{t+1}$$

trong đó, V_i^{t+1} là tốc độ của particle *i* tại thế hệ thứ $t + 1$ và được cập nhật bởi:

$$V_i^{t+1} = \omega \times V_i^t + c1 \times r1(P_i^t - X_i^t) + c2 \times r2(P_g^t - X_i^t)$$

trong đó, P_i^t và P_g^t là phương án **tốt nhất cục bộ** và **tốt nhất toàn cục** trong swarm được tìm thấy cho tới thế hệ thứ *t*

r1, r2 là hai số ngẫu nhiên trong [0, 1]

Tìm kiếm tối ưu theo kinh nghiệm

- Thuật toán tối ưu bầy đàn
- **Bước 1**: Khởi tạo biến lặp *t*, sinh ngẫu nhiên swarm *S* trong không gian tìm kiếm
- **Bước 2**: Tính toán giá trị hàm mục tiêu $f(x_i)$ cho tất cả các particle
- **Bước 3**: Cập nhật phương án cục bộ tốt nhất P_i^t cho tất cả các particle
- **Bước 4**: Cập nhật phương án tốt nhất trong swarm P_g^t
- **Bước 5**: Tính tốc độ V_i^{t+1} cho các particle
- **Bước 6**: Di chuyển các particle tới vị trí mới X_i^{t+1}
- **Bước 7**: Tăng giá trị biến lặp *t*. Nhảy tới **bước 2** và lặp lại cho tới khi hội tụ hoặc đạt giá trị lớn nhất của *t*

Tìm kiếm tối ưu theo kinh nghiệm

- Thuật toán tối ưu bầy đàn
  - Ví dụ 1: Giải phương trình

    $3x + 2y + z + q = 34$
  - Các tham số:
    - Kích thước quần thể (số particle): 10
    - Số thế hệ (số lần lặp): 100
    - Các hệ số $c1 = c2 = 1.5$
    - Hệ số Inertia $w = 0.7$
    - Ràng buộc: 1 <= x, y, z, q <= 20
    - Hàm mục tiêu |f(x) - 34| → 0 (cực tiểu hóa)

Tìm kiếm tối ưu theo kinh nghiệm

- Thuật toán tối ưu bầy đàn
  - Ví dụ 2: Banana function

    Cực đại hóa hàm $f(x) = 2x – x^2/16$, $x \in [0, 31]$
  - Các tham số:
    - Kích thước quần thể (số particle): 4
    - Số thế hệ (số lần lặp): 2
    - Các hệ số c1 = c2 = 2

Tìm kiếm tối ưu theo kinh nghiệm

- Thuật toán tối ưu bầy đàn
  - Bước khởi tạo
    - Vị trí và tốc độ (velocity) của các particle được khởi tạo ngẫu nhiên trong khoảng (0, 31)

Particle (x)	Velocity (vel)	Fitness
18.663	12.69	15.556
28.574	0.0106	9.01
30.639	16.7689	2.605
28.816	6.444	5.732

    - Ta có, Max fitness: **15.556** tại vị trí của particle **18.663**
    - Ký hiệu Gb là vị trí của particle có giá trị hàm fitness lớn nhất toàn cục, Pb là vị trí tốt nhất cục bộ

Tìm kiếm tối ưu theo kinh nghiệm

- Thuật toán tối ưu bầy đàn
  - Tạo swarm mới, t = 1
    - Sinh ngẫu nhiên các số r1 và r2

r1	r2
0.2193	0.7485
0.325	0.5433
0.095	0.3381
0.745	0.8323

    - Tính velocity cho các particle: ω = max(t)/t = 2/1 = 2

      vel = ω * vel + c1 * r1 * (Pb - x) + c2 * r2 (Gb - x)
    - Tính vị trí mới cho các particle

      x = x + vel

Tìm kiếm tối ưu theo kinh nghiệm

- Thuật toán tối ưu bầy đàn
  - Giá trị của các vị trí mới

Particle (x)
25.011
17.983
30.924
15.1346

Chú ý: kiểm tra xem x có thuộc [0, 31] hay không?

New fitness
10.924
15.7541
2.07
15.953

So sánh với local fitness để chọn giá trị tốt nhất

Local fitness
15.556
9.0107
2.6050
5.7329

Tìm kiếm tối ưu theo kinh nghiệm

- Thuật toán tối ưu bầy đàn
  - Chọn được các particle với fitness tốt hơn

Better fitness	Better particle
15.556	18.663
15.7541	17.983
2.6050	30.63
15.9532	15.134

- Hiện tại: Gb = 15.1346 (vị trí của particle tốt nhất)
- Max fitness = 15.953
- Local fitness = better fitness
- Local particle (Pb) = better particle

Tìm kiếm tối ưu theo kinh nghiệm

- Thuật toán tối ưu bầy đàn
  - Sau thế hệ thứ nhất (vòng lặp 1) ta có vị trí tốt nhất Gb = 15.1346 và Max fitness = 15.953
  - Tiếp theo tạo swarm mới, t = 2
    - Sinh ngẫu nhiên các số r1 và r2

r1	r2
0.5529	0.5464
0.9575	0.3967
0.892	0.6228
0.356	0.7960

    - Tính velocity cho các particle: ω = max(t)/t = 2/2 = 1

      vel = ω * vel + c1 * r1 * (Pb - x) + c2 * r2 (Gb - x)
    - Tính vị trí mới cho các particle

      x = x + vel

Tìm kiếm tối ưu theo kinh nghiệm

- Thuật toán tối ưu bầy đàn
  - Tính giá trị velocity (vel) và vị trí mới cho các cá thể

vel	Particle (x)
17.8	7.2024
1.9	16.00
20.177	10.7471
0	15.1346

New fitness
11.16
16.00
14.27
15.9

So sánh với local fitness để chọn giá trị tốt nhất

Local fitness
15.556
5.7541
2.6
15.953

Tìm kiếm tối ưu theo kinh nghiệm

- Thuật toán tối ưu bầy đàn
  - Chọn được các particle với fitness tốt hơn

Better fitness	Better particle (x)
15.556	18.663
16.00	16.00
14.27	10.7471
15.953	15.1346

    - Hiện tại: Gb = 16 (vị trí của particle tốt nhất)
    - Max fitness = 16
    - Local fitness = better fitness
    - Local particle (Pb) = better particle
  - **Sau vòng lặp thứ hai: Gb = 16 và Max fitness = 16**

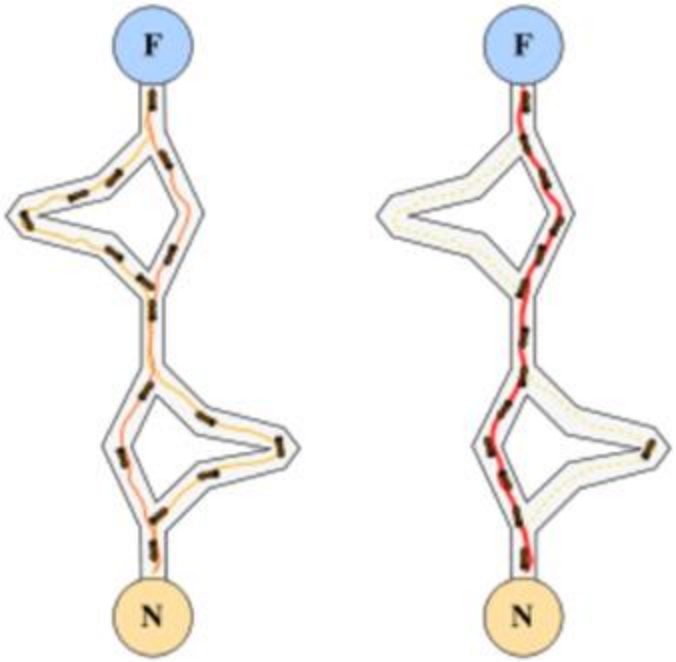
Tìm kiếm tối ưu theo kinh nghiệm

- Thuật toán tối ưu bầy đàn
  - **Ưu điểm**
    - Không nhạy cảm với việc mở rộng quy mô của các biến thiết kế
    - Cài đặt đơn giản
    - Dễ dàng song song hóa
    - Sử dụng ít tham số
    - Rất hiệu quả trong tìm kiếm tối ưu toàn cục
  - **Nhược điểm**
    - Hội tụ chậm trong giai đoạn tìm kiếm tinh chỉnh (khả năng tìm kiếm địa phương yếu)

PSO được áp dụng hiệu quả vào giải quyết nhiều bài toán thực tế

Tìm kiếm tối ưu theo kinh nghiệm

- Thuật toán tối ưu đàn kiến (ACO)
  - Tối ưu đàn kiến (Ant Colony Optimization - ACO) được Marco Dorigo giới thiệu năm 1992
    - Là một kỹ thuật xác suất
    - Tìm kiếm đường đi tối ưu trong đồ thị dựa trên hành vi tìm kiếm đường đi của kiến từ tổ của chúng cho đến nguồn thức ăn
    - Tối ưu hóa Meta-heuristic



Tìm kiếm tối ưu theo kinh nghiệm

- Thuật toán tối ưu đàn kiến (ACO)
  - Một số khái niệm
    - Kiến điều hướng từ nơi tổ của chúng tới nguồn thức ăn. Chúng định hướng bằng **vết mùi** mà chúng để lại (pheromone)
    - Mỗi con kiến đều di chuyển một cách ngẫu nhiên
    - Vết mùi được kiến lưu lại trên toàn bộ quãng đường đi của chúng, vết mùi bị bay hơi theo thời gian
    - Đoạn đường nào có nhiều vết mùi hơn sẽ được đi theo nhiều hơn
    - Đường đi ngắn nhất được phát hiện thông qua **vết mùi** mà chúng đã để lại trên đường đi

Tìm kiếm tối ưu theo kinh nghiệm

- Thuật toán tối ưu đàn kiến (ACO)
  - Thuật toán ACO

```
Khởi tạo: ma trận vết mùi, số kiến n_ants
while (điều dừng chưa thỏa) {
    for (i = 0; i < n_ants; i++) {
        Xây dựng lời giải;
        Cập nhật lời giải tốt;
    }
    Cập nhật vết mùi;
}
```

Tìm kiếm tối ưu theo kinh nghiệm

- Thuật toán tối ưu đàn kiến (ACO)
  - Khi áp dụng phương pháp ACO cho các bài toán cụ thể, ba yếu tố sau có ảnh hưởng quyết định đến hiệu quả thuật toán
    - Xây dựng đồ thị cấu trúc thích hợp → giảm miền tìm kiếm của kiến
    - Chọn thông tin heuristic tốt → tăng hiệu quả thuật toán
    - Chọn quy tắc cập nhật mùi: cập nhật mùi địa phương (AS), cập nhật mùi toàn cục (ACS), MMAS, SMMAS

Bài tập

- Bài toán:

  Giải phương trình: $3x + 2y + z + q = 34$

- Viết chương trình bằng ngôn ngữ C++ và Python giải bài toán trên sử dụng:
  - Thuật toán di truyền
  - Thuật toán PSO

TỔNG KẾT

1. **Tổng quan**