Sự kỳ diệu của ĐỆ QUI và TRUY VẾT

Đỗ Phan Thuận

Email/Facebook: thuandp.sinhvien@gmail.com
Khoa Học Máy Tính
Trường Đại Học Bách Khoa Hà Nội.

Ngày 16 tháng 5 năm 2022

Các mô hình giải bài cơ bản

Các phương pháp căn bản xây dựng lời giải bài toán

- Duyệt toàn bộ
- Chia để trị
- Quy hoạch động
- Tham lam

Mỗi mô hình ứng dụng cho nhiều loại bài toán khác nhau

Đệ qui là gì

Đệ qui là gì

Trong thực tế ta thường gặp những đối tượng bao gồm chính nó hoặc được định nghĩa dưới dạng của chính nó. Ta nói các đối tượng đó được xác định một cách đệ qui

Đệ qui và qui nạp

Đệ qui và qui nạp toán học có những nét tương đồng và là bổ sung cho nhau. Định nghĩa đệ qui thường giúp cho chứng minh bằng qui nạp các tính chất của các đối tượng được định nghĩa đệ qui. Ngược lại, các chứng minh bằng qui nặp toán học thường là cơ sở để xây dựng các thuật toán đệ qui để giải quyết nhiều bài toán:

(1) Bước cơ sở qui nạp —> giống như bước cơ sở trong định nghĩa đệ qui

(2) Bước chuyển qui nạp —> giống như bước đệ qui

Kỹ thuật đệ qui

Kỹ thuật đệ qui là kỹ thuật tự gọi đến chính mình với đầu vào kích thước thường là nhỏ hơn

Việc phát triển kỹ thuật đệ qui là thuận tiện khi cần xử lý với các đối tượng được định nghĩa đệ qui (chẳng hạn: tập hợp, hàm, cây, . . .)

Các mô hình giải bài cơ bản

Các phương pháp căn bản xây dựng lời giải bài toán:

- Duyệt toàn bộ
- Chia để trị
- Quy hoạch động
- Tham lam

Các mô hình giải bài cơ bản

Các phương pháp căn bản xây dựng lời giải bài toán:

- Duyệt toàn bộ

Duyệt toàn bộ đa phần phải sử dụng kỹ thuật đệ qui
(Một phương pháp ít phổ biến hơn là phương pháp sinh kế tiếp)

- Chia để trị
- Quy hoạch động
- Tham lam

Các mô hình giải bài cơ bản

Các phương pháp căn bản xây dựng lời giải bài toán:

- Duyệt toàn bộ
- Chia để trị
- Quy hoạch động
- Tham lam

Các mô hình giải bài cơ bản

Các phương pháp căn bản xây dựng lời giải bài toán:

- Duyệt toàn bộ

Duyệt toàn bộ đa phần phải sử dụng kỹ thuật đệ qui
(Một phương pháp ít phổ biến hơn là phương pháp sinh kế tiếp)

- Chia để trị

Các thuật toán được phát triển dựa trên phương pháp chia để trị thông thường được mô tả dưới dạng kỹ thuật đệ qui

- Quy hoạch động
- Tham lam

Các mô hình giải bài cơ bản

Các phương pháp căn bản xây dựng lời giải bài toán:

- Duyệt toàn bộ
- Chia để trị
- Quy hoạch động
- Tham lam

Các mô hình giải bài cơ bản

Các phương pháp căn bản xây dựng lời giải bài toán:

- Duyệt toàn bộ

Duyệt toàn bộ đa phần phải sử dụng kỹ thuật đệ qui
(Một phương pháp ít phổ biến hơn là phương pháp sinh kế tiếp)

- Chia để trị

Các thuật toán được phát triển dựa trên phương pháp chia để trị thông thường được mô tả dưới dạng kỹ thuật đệ qui

- Quy hoạch động

Các thuật toán được phát triển dựa trên phương pháp qui hoạch động trở nên sáng sủa hơn khi được mô tả dưới dạng kỹ thuật đệ qui

- Tham lam

Các mô hình giải bài cơ bản

Các phương pháp căn bản xây dựng lời giải bài toán:

- Duyệt toàn bộ

Duyệt toàn bộ đa phần phải sử dụng kỹ thuật đệ qui
(Một phương pháp ít phổ biến hơn là phương pháp sinh kế tiếp)

- Chia để trị

Các thuật toán được phát triển dựa trên phương pháp chia để trị thông thường được mô tả dưới dạng kỹ thuật đệ qui

- Quy hoạch động

Các thuật toán được phát triển dựa trên phương pháp qui hoạch động trở nên sáng sủa hơn khi được mô tả dưới dạng kỹ thuật đệ qui

- Tham lam

Các thuật toán tham có thể cài đặt theo phương pháp đệ qui

Mô hình chung của đệ qui

```
void TRY(input) {
  if (Input size small enough) {
    do Base_case
       //giai bai toan kich thuoc dau vao nho nhat
  } else { //buoc de qui
      [Trong cong thuc de qui,
      voi moi kich thuoc nho hon cua bai toan con]
         call TRY(new input);
      [To hop loi giai cua cac bai toan con]
      return solution;
  }
}
```

Các mô hình giải bài cơ bản

Các phương pháp căn bản xây dựng lời giải bài toán

- Duyệt toàn bộ
- Chia để trị
- Quy hoạch động
- Tham lam

Mỗi mô hình ứng dụng cho nhiều loại bài toán khác nhau

Chia để trị

Chia để trị là một mô hình giải bài theo hướng làm dễ bài toán đi bằng cách chia thành các phần nhỏ hơn, độc lập và xử lý từng phần một

Thông thường làm theo 3 bước chính:

1. CHIA: chia bài toán thành một hay nhiều bài toán con - thường hay chia một nửa hoặc gần một nửa
2. XỬ LÝ: giải đệ quy mỗi bài toán con - mỗi bài toán cần giải trở nên dễ hơn
3. KẾT HỢP: kết hợp lời giải các bài toán con thành lời giải bài toán ban đầu

Mô hình chung của chia để trị

```
void  DC(n) {
  if n <= n0 {
     [Giai bai toan con mot cach truc tiep]
  } else {
     [Chia bai toan thanh a bai toan con kich thuoc n/b]
     [Voi moi bai toan trong a bai toan con] {
        call DC(n/b)
     }
     [Tong hop loi giai cua a bai toan con]
     return solution;
  }
}
```

- n0 là kích thước nhỏ nhất của bài toán con (còn gọi là neo đệ qui). Bài toán con với kích thước n0 sẽ được giải trực tiếp
- a là số lượng bài toán con cần giải
- b liên quan đến kích thước của bài toán con được chia

Các mô hình giải bài cơ bản

Các phương pháp căn bản xây dựng lời giải bài toán

- Duyệt toàn bộ
- Chia để trị
- Quy hoạch động
- Tham lam

Mỗi mô hình ứng dụng cho nhiều loại bài toán khác nhau

RICHARD BELLMAN ON THE BIRTH OF DYNAMIC PROGRAMMING

STUART DREYFUS

University of California, Berkeley, IEOR, Berkeley, California 94720, dreyfus@ieor.berkeley.edu

What follows concerns events from the summer of 1949, when Richard Bellman first became interested in multistage decision problems, until 1955. Although Bellman died on March 19, 1984, the story will be told in his own words since he left behind an entertaining and informative autobiography, *Eye of the Hurricane* (World Scientific Publishing Company, Singapore, 1984), whose publisher has generously approved extensive excerpting.

During the summer of 1949 Bellman, a tenured associate professor of mathematics at Stanford University with a developing interest in analytic number theory, was consulting for the second summer at the RAND Corporation in Santa Monica. He had received his Ph.D. from Princeton in 1946 at the age of 25, despite various war-related activities during World War II—including being assigned by the Army to the Manhattan Project in Los Alamos. He had already exhibited outstanding ability both in pure mathematics and in solving applied problems arising from the physical world. Assured of a successful conventional academic career, Bellman, during the period under consideration, cast his lot instead with the kind of applied mathematics later to be known as operations research. In those days applied practitioners were regarded as distinctly second-class citizens of the mathematical fraternity. Always one to enjoy controversy, when invited to speak at various university mathematics department seminars, Bellman delighted in justifying his choice of applied over pure mathematics as being motivated by the real world's greater challenges and mathematical demands.

what RAND was interested in. He suggested that I work on multistage decision processes. I started following that suggestion" (p. 157).

CHOICE OF THE NAME DYNAMIC PROGRAMMING

"I spent the Fall quarter (of 1950) at RAND. My first task was to find a name for multistage decision processes.

"An interesting question is, 'Where did the name, dynamic programming, come from?' The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word, research. I'm not using the term lightly; I'm using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term, research, in his presence. You can imagine how he felt, then, about the term, mathematical. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose? In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word, 'programming.' I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying—I thought, let's kill two birds with one stone. Let's take a word that has an

Hình: R.E.Bellman (1920-1984)

Trong chiến tranh thế giới thứ 2, các ngành khoa học cơ bản ở Mỹ không được đầu tư để dành toàn bộ nguồn lực cho thế chiến, chỉ những kết quả khoa học ứng dụng trực tiếp cho chiến trường mới được cấp kinh phí nghiên cứu, ví dụ: qui hoạch tuyến tính với bài toán khẩu phần ăn cho binh sĩ.

Nhà toán học Bellman thời kỳ đó nghiên cứu ra phương pháp 'multisage deision processes' (quá trình ra quyết định thông qua nhiều lớp) trong lĩnh vực lập kế hoạch (planning). Tuy nhiên từ 'planning' không phù hợp vào thời kỳ đó nên ông đã thay bằng từ 'programming' (lập trình) thời thượng hơn khi mà máy tính to đầu tiên của quân đội Mỹ ra đời. Tiếp theo ông thay từ 'multisage' bằng từ 'dynamic' nghe hay hơn thể hiện sự gối nhau về thời gian. Thuật ngữ 'dynamic programming' ra đời từ đó.

Dynamic programming mang tính kỹ thuật lập trình nhiều hơn là tính mô hình dạng bài toán (như quy hoạch tuyến tính), tuy nhiên từ dịch ra 'Quy Hoạch Động' nghe hay và thuận hơn từ 'Lập Trình Động'.

Qui hoạch động là gì?

- Là một mô hình giải bài
- Nhiều điểm tương đồng với hai phương pháp Chia để trị và Quay lui
- Nhắc lại Chia để trị:
  - Chia bài toán cha thành các bài toán con *độc lập*
  - Giải từng bài toán con (bằng đệ qui)
  - Kết hợp lời giải các bài toán con lại thành lời giải của bài toán cha
- Phương pháp qui hoạch động:
  - Chia bài toán cha thành các bài toán con *gối nhau*
  - Giải từng bài toán con (bằng đệ qui)
  - Kết hợp lời giải các bài toán con lại thành lời giải của bài toán cha
  - *Không tìm nhiều hơn một lần lời giải của cùng một bài toán*

Công thức Qui hoạch động

1. Tìm công thức qui hoạch động cho bài toán dựa trên các bài toán con
2. Cài đặt công thức qui hoạch động: Chuyển công thức thành hàm đệ qui
3. Lưu trữ kết quả các hàm đã tính toán

Nhận xét

Bước 1: tìm công thức qui hoạch động là bước khó nhất và quan trọng nhất. Bước 2 và 3 có thể áp dụng sơ đồ chung sau đây để thực hiện

Hàm đệ qui

```
map<problem, value> memory;

value dp(problem P) {
    if (is_base_case(P)) {
        return base_case_value(P);
    }

    if (memory.find(P) != memory.end()) {
        return memory[P];
    }

    value result = some value;
    for (problem Q in subproblems(P)) {
        result = combine(result, dp(Q));
    }

    memory[P] = result;
    return result;
}
```

Bình luận

- Việc sử dụng hàm đệ qui để cài đặt công thức qui hoạch động là cách tiếp cận lập trình tự nhiên và đơn giản cho lập trình giải bài toán qui hoạch động, ta gọi đó là cách tiếp cận lập trình Top-Down, phù hợp với đa số người mới tiếp cận kỹ thuật Qui hoạch động và phù hợp với những bài toán giải thuần tuý dựa vào công thức qui hoạch động
- Khi đã quen thuộc với các bài qui hoạch động ta có thể luyện tập phương pháp lập trình Bottom-Up, xây dựng dần lời giải từ các bài toán con đến các bài toán cha. Phương pháp Bottom-Up sẽ giúp việc áp dụng các cấu trúc dữ liệu nâng cao linh hoạt hơn nhằm cải tiến độ phức tạp trong các bài toán qui hoạch động, ví dụ: cây IT/BIT, hàng đợi ưu tiên, hàng đợi hai đầu (Deque), ...
- Các bước trên mới chỉ tìm ra được giá trị tối ưu của bài toán. Nếu phải đưa ra các phần tử trong lời giải tạo nên giá trị tối ưu của bài toán thì cần thực hiện thêm bước Truy vết. Bước Truy vết nên mô phỏng lại Bước 2 cài đặt đệ qui và tìm ra các phần tử của lời giải dựa trên thông tin các bài toán con đã được lưu trữ trong mảng memmory

Dãy Fibonacci

Hai số đầu tiên của dãy Fibonacci là 1 và 1. Tất cả các số khác của dãy được tính bằng tổng của hai số ngay trước nó trong dãy

- Yêu cầu: Tìm số Fibonacci thứ n
- Thử giải bài toán bằng phương pháp Qui hoạch động

1. Tìm công thức truy hồi:

$$\begin{aligned}
\text{fibonacci}(1) &= 1 \\
\text{fibonacci}(2) &= 1 \\
\text{fibonacci}(n) &= \text{fibonacci}(n-2) + \text{fibonacci}(n-1)
\end{aligned}$$

Dãy Fibonacci

2. Cài đặt công thức qui hoạch động

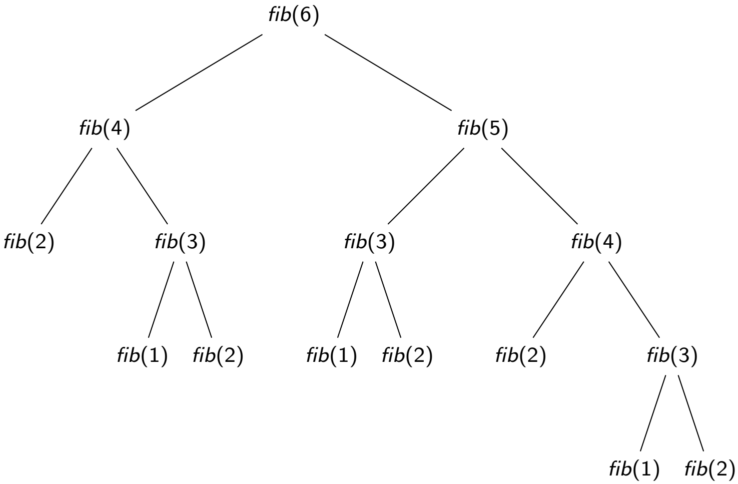
```
int fibonacci(int n) {
    if (n <= 2) {
        return 1;
    }

    int res = fibonacci(n - 2) + fibonacci(n - 1);

    return res;
}
```

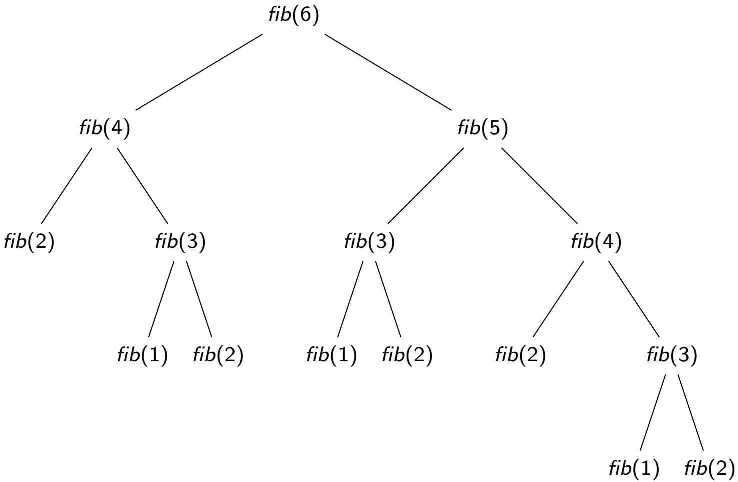
Dãy Fibonacci

- Độ phức tạp là bao nhiêu?



Dãy Fibonacci

- Độ phức tạp là bao nhiêu? Hàm mũ, gần như $O(2^n)$



Dãy Fibonacci

3. Lưu trữ kết quả các hàm đã tính

```
map<int, int> mem;

int fibonacci(int n) {
    if (n <= 2) {
        return 1;
    }

    if (mem.find(n) != mem.end()) {
        return mem[n];
    }

    int res = fibonacci(n - 2) + fibonacci(n - 1);

    mem[n] = res;
    return res;
}
```

Dãy Fibonacci

```
int mem[1000];
for (int i = 0; i < 1000; i++)
    mem[i] = -1;

int fibonacci(int n) {
    if (n <= 2) {
        return 1;
    }
    if (mem[n] != -1) {
        return mem[n];
    }
    int res = fibonacci(n - 2) + fibonacci(n - 1);

    mem[n] = res;
    return res;
}
```

Bây giờ độ phức tạp là bao nhiêu?

Dãy Fibonacci

- Ta có n khả năng input cho hàm đệ qui: 1, 2, ..., n.
- Với mỗi input:
  - hoặc là kết quả được tính và lưu trữ lại
  - hoặc là lấy luôn ra từ bộ nhớ nếu như trước đây đã được tính
- Mỗi input sẽ được tính tốt đa một lần
- Thời gian tính là $O(n \times f)$, với f là thời gian tính toán của hàm với một input, với giả thiết là kết quả đã tính trước đây sẽ được lấy trực tiếp từ bộ nhớ, chỉ trong $O(1)$
- Do ta chỉ tốn một lượng hằng số phép tính đối với một input của hàm, nên $f = O(1)$
- Thời gian tính tổng cộng là $O(n)$

Tổng lớn nhất trong mảng

- Cho một mảng số nguyên $\mathrm{arr}[0], \mathrm{arr}[1], \ldots, \mathrm{arr}[n-1]$, hãy tìm một đoạn trong mảng có trọng số lớn nhất, nghĩa là tổng các số trong đoạn là lớn nhất

-15	8	-2	1	0	6	-3

Tổng lớn nhất trong mảng

- Cho một mảng số nguyên $\mathrm{arr}[0], \mathrm{arr}[1], \ldots, \mathrm{arr}[n-1]$, hãy tìm một đoạn trong mảng có trọng số lớn nhất, nghĩa là tổng các số trong đoạn là lớn nhất

-15	8	-2	1	0	6	-3

- Tổng của đoạn có trọng số lớn nhất trong mảng là 13

Tổng lớn nhất trong mảng

- Cho một mảng số nguyên $\mathrm{arr}[0], \mathrm{arr}[1], \ldots, \mathrm{arr}[n-1]$, hãy tìm một đoạn trong mảng có trọng số lớn nhất, nghĩa là tổng các số trong đoạn là lớn nhất

-15	8	-2	1	0	6	-3

- Tổng của đoạn có trọng số lớn nhất trong mảng là 13
- Cách giải thế nào?
  - Phương pháp trực tiếp thử tất cả gần $\approx n^2$ khoảng, và tính trọng số mỗi đoạn, cho độ phức tạp$O(n^3)$
  - Ta có thể xử lý kỹ thuật bởi một "mẹo" lưu trữ cố định trong vòng lặp để giảm độ phức tạp về $O(n^2)$
  - Liệu có thể làm tốt hơn với phương pháp Qui hoạch động?

Tổng lớn nhất trong mảng

- Bước đầu tiên là đi tìm công thức qui hoạch động
- Gọi $\mathrm{max\_sum}(i)$ là trọng số của đoạn có trọng số lớn nhất giới hạn trong đoạn $0, \ldots, i$
- Bước cơ sở: $\mathrm{max\_sum}(0) = \max(0, arr[0])$
- $\mathrm{max\_sum}(i)$?
- Liên hệ gì với $\mathrm{max\_sum}(i-1)$?
- Liệu có thể kết hợp lời giải của các bài toán con có kích thước bé hơn i thành lời giải bài toán có kích thước bằng i?
- Không hoàn toàn hiển nhiên phải không ?...

Tổng lớn nhất trong mảng

- Hãy thay đổi hàm mục tiêu:
- Gọi $\mathrm{max\_sum}(i)$ là trọng số đoạn có trọng số lớn nhất giới hạn bởi $0, \ldots, i$, *mà phải kết thúc tại i*
- Bước cơ sở: $\mathrm{max\_sum}(0) = arr[0]$
- $\mathrm{max\_sum}(i) = \max(arr[i], arr[i] + \mathrm{max\_sum}(i-1))$
- Vậy công thức cuối cùng chỉ là $\max_{0 \leq i < n} \{ \mathrm{max\_sum}(i) \}$

Tổng lớn nhất trong mảng

- Bước tiếp theo là cài đặt công thức qui hoạch động

```
int arr[1000];

int max_sum(int i) {
    if (i == 0) {
        return arr[i];
    }

    int res = max(arr[i], arr[i] + max_sum(i - 1));
    return res;
}
```

Tổng lớn nhất trong mảng

- Bước cuối cùng là lưu trữ các hàm đã tính

```
int arr[1000];
int mem[1000];
bool comp[1000];
memset(comp, 0, sizeof(comp));

int max_sum(int i) {
    if (i == 0) {
        return arr[i];
    }
    if (comp[i]) {
        return mem[i];
    }

    int res = max(arr[i], arr[i] + max_sum(i - 1));
    mem[i] = res;
    comp[i] = true;
    return res;
}
```

Tổng lớn nhất trong mảng

- Trong thủ tục chính chỉ cần gọi đệ qui một lần cho $\mathrm{max\_sum}(n-1)$, hàm đệ qui sẽ tiến hành tính toàn bộ các giá trị của $\mathrm{max\_sum}(i), 0 \leq i \leq n-1$
- Kết quả bài toán đơn giản là giá trị lớn nhất trong các giá trị $\mathrm{max\_sum}(i)$ đã được lưu trữ trong $\mathrm{mem}[i]$ sau quá trình gọi đệ qui

```
int maximum = 0;
for (int i = 0; i < n; i++) {
    maximum = max(maximum, mem[i]);
}

printf("%d\n", maximum);
```

- Lưu ý nếu bài toán yêu cầu tìm đoạn có trọng số lớn nhất trong nhiều mảng khác nhau, thì hãy nhớ xóa bộ nhớ khi kết thúc tính toán mỗi mảng

Tổng lớn nhất trong mảng

- Độ phức tạp tính toán ?

Tổng lớn nhất trong mảng

- Độ phức tạp tính toán ?
- Có n khả năng input cho hàm đệ qui
- Mỗi input được tính trong $O(1)$, giả thiết là mỗi phép gọi đệ qui là $O(1)$
- Thời gian tính toán tổng cộng là $O(n)$

Tổng lớn nhất trong mảng

- Độ phức tạp tính toán ?
- Có n khả năng input cho hàm đệ qui
- Mỗi input được tính trong $O(1)$, giả thiết là mỗi phép gọi đệ qui là $O(1)$
- Thời gian tính toán tổng cộng là $O(n)$
- Làm thế nào để biết chính xác một đoạn nào trong mảng tạo ra giá trị tổng lớn nhất tìm được?

Tổng lớn nhất trong mảng - Truy vết bằng đệ qui

```
void trace(int i) {
    if (i != 0 &&
            mem[i] == arr[i] + mem[i-1]) {
        trace(i - 1);
    }
    printf("%d ", arr[i]);
}
```

Tổng lớn nhất trong mảng - Truy vết bằng vòng lặp

```
int maximum = 0, pos = -1;
for (int i = 0; i < n; i++) {
    maximum = max(maximum, mem[i]);
    if (maximum == mem[i]) pos = i;
}

printf("%d\n", maximum);
int L = pos, R = pos, sum = arr[L];
while (sum != maximum){
    --L;
    sum += arr[L];
}
printf("%d %d", L, R);
```

Đổi tiền

- Cho trước một tập các đồng tiền mệnh giá $d_0, d_1, \ldots, d_{n-1}$, và một mệnh giá x. Hãy tìm số lượng ít nhất các đồng tiền để đổi cho mệnh giá x?
- Có ai biết thuật toán tham lam cho bài Đổi tiền này?
- Lời giải thuật toán tham lam không hề chắc chắn đưa ra lời giải tối ưu, thậm chí nhiều trường hợp còn không đưa ra được lời giải...

- Có thể sử dụng phương pháp Qui hoạch động?

Đổi tiền

- Bước đầu tiên: xây dựng công thức Qui hoạch động

- Gọi $\mathrm{opt}(i, x)$ là số lượng tiền ít nhất cần để đổi mệnh giá x nếu chỉ được phép sử dụng các đồng tiền mệnh giá $d_0, \ldots, d_i$

- Bước cơ sở: $\mathrm{opt}(i, x) = \infty$ nếu $x < 0$
- Bước cơ sở: $\mathrm{opt}(i, 0) = 0$
- Bước cơ sở: $\mathrm{opt}(-1, x) = \infty$

- $\mathrm{opt}(i, x) = \min \begin{cases} 1 + \mathrm{opt}(i, x - d_i) \\ \mathrm{opt}(i - 1, x) \end{cases}$

Đổi tiền

```
int INF = 100000;
int d[10];

int opt(int i, int x) {
    if (x < 0) return INF;
    if (x == 0) return 0;
    if (i == -1) return INF;

    int res = INF;
    res = min(res, 1 + opt(i, x - d[i]));
    res = min(res, opt(i - 1, x));

    return res;
}
```

Đổi tiền

```
int INF = 100000;
int d[10];
int mem[10][10000];
memset(mem, -1, sizeof(mem));

int opt(int i, int x) {
    if (x < 0) return INF;
    if (x == 0) return 0;
    if (i == -1) return INF;

    if (mem[i][x] != -1) return mem[i][x];

    int res = INF;
    res = min(res, 1 + opt(i, x - d[i]));
    res = min(res, opt(i - 1, x));

    mem[i][x] = res;
    return res;
}
```

Đổi tiền

- Độ phức tạp?

Đổi tiền

- Độ phức tạp?
- Số lượng khả năng input là $n \times x$
- Mỗi input được xử lý trong $O(1)$, giả thiết mỗi lời gọi đệ qui thực hiện trong thời gian hằng số
- Thời gian tính toán tổng cộng là $O(n \times x)$

Đổi tiền

- Độ phức tạp?
- Số lượng khả năng input là $n \times x$
- Mỗi input được xử lý trong $O(1)$, giả thiết mỗi lời gọi đệ qui thực hiện trong thời gian hằng số
- Thời gian tính toán tổng cộng là $O(n \times x)$
- Làm thế nào để xác định được những đồng tiền nào cho phương án tối ưu ?
- Hãy truy vết ngược lại quá trình đệ qui

Đổi tiền - Truy vết bằng đệ qui

```
void trace(int i, int x) {
    if (x < 0) return;
    if (x == 0) return;
    if (i == -1) return;

    int res = INF;
    if (mem[i][x] == 1 + mem[i][x - d[i]]){
        printf("%d ", d[i]);
        trace(i, x - d[i]);
    } else {
        trace(i-1, x);
    }
}
```

Đổi tiền - Truy vết bằng vòng lặp

```
int answer = mem[n-1][x];
printf("%d\n", answer);
for (int i = n-1, k = 0; k < answer; ++k) {
    if (mem[i][x] == 1 + mem[i][x-d[i]]){
        printf("%d ", d[i]);
        x -= d[i];
    } else {
        --i;
    }
}
```

Dãy con chung dài nhất

- Cho hai xâu (hoặc hai mảng số nguyên) n phần tử $a[0], \ldots, a[n-1]$ và $b[0], \ldots, b[m-1]$, hãy tìm độ dài của dãy con chung dài nhất của hai xâu.

- $a =$"b<u>a</u><u>n</u>a<u>n</u><u>i</u><u>n</u><u>n</u>"
- $b =$"k<u>a</u><u>n</u><u>i</u><u>n</u>a<u>n</u>"

- Dãy con chung dài nhất của a và b, "aninn", có độ dài 5

Dãy con chung dài nhất

- Gọi $\mathrm{lcs}(i, j)$ là độ dài dãy con chung dài nhất của $a[0], \ldots, a[i]$ và $b[0], \ldots, b[j]$

- Bước cơ sở: $\mathrm{lcs}(-1, j) = 0$
- Bước cơ sở: $\mathrm{lcs}(i, -1) = 0$

- $\mathrm{lcs}(i, j) = \max \begin{cases} \mathrm{lcs}(i, j-1) \\ \mathrm{lcs}(i-1, j) \\ 1 + \mathrm{lcs}(i-1, j-1) & \text{nếu } a[i] = b[j] \end{cases}$

Dãy con chung dài nhất

```
string a = "bananinn",
       b = "kaninan";
int mem[1000][1000];
memset(mem, -1, sizeof(mem));

int lcs(int i, int j) {
    if (i == -1 || j == -1) {
        return 0;
    }
    if (mem[i][j] != -1) {
        return mem[i][j];
    }
    int res = 0;
    res = max(res, lcs(i, j - 1));
    res = max(res, lcs(i - 1, j));
    if (a[i] == b[j]) {
        res = max(res, 1 + lcs(i - 1, j - 1));
    }
    mem[i][j] = res;
    return res;
}
```

Dãy con chung dài nhất

- Độ phức tạp tính toán?

Dãy con chung dài nhất

- Độ phức tạp tính toán?

- Có n khả năng input
- Mỗi input được tính trong thời gian $O(1)$, giả thiết mỗi lời gọi đệ qui chỉ mất $O(1)$
- Thời gian tính tổng cộng là $O(n \times m)$

Dãy con chung dài nhất

- Độ phức tạp tính toán?

- Có n khả năng input
- Mỗi input được tính trong thời gian $O(1)$, giả thiết mỗi lời gọi đệ qui chỉ mất $O(1)$
- Thời gian tính tổng cộng là $O(n \times m)$
- Làm thế nào để biết chính xác những phần tử nào thuộc dãy con tăng dài nhất?

Dãy con chung dài nhất - Truy vết bằng đệ qui

```
void trace(int i, int j) {
    if (i == -1 || j == -1) {
        return;
    }
    if (mem[i][j] == mem[i-1][j]){
        trace(i-1, j);
        return;
    }
    if (mem[i][j] == mem[i][j-1]){
        trace(i, j-1);
        return;
    }
    if (a[i] == b[j] && mem[i][j] == 1 + mem[i-1][j-1]){
        trace(i-1, j-1);
        printf("%d ", a[i]);
        return;
    }
}
```

Dãy con chung dài nhất - Truy vết bằng vòng lặp

```
int answer = lcs(n-1, n-1);
printf("%d\n", answer);
stack<int> s;
for (int i = n-1, j = n-1, k = 0; k < answer; ++k) {
    if (a[i] == b[j] && mem[i][j] == 1 + mem[i-1][j-1]){
        s.push(a[i]);
        --i;
        --j; continue;
    }
    if (mem[i][j] == mem[i-1][j]){
        --i; continue;
    }
    if (mem[i][j] == mem[i][j-1]){
        --j; continue;
    }
}
while (!s.empty()) {
    printf("%d ", s.back());
    s.pop();
}
```

Dãy con tăng dài nhất

- Cho một mảng n số nguyên $a[0], a[1], \ldots, a[n-1]$, hãy tìm độ dài của dãy con tăng dài nhất?

- Định nghĩa dãy con?
- Nếu xoá đi 0 phần tử hoặc một số phần tử của mảng a thì sẽ thu được một dãy con của a

- Ví dụ: $a = [5, 1, 8, 1, 9, 2]$

- $[5, 8, 9]$ là một dãy con
- $[1, 1]$ là một dãy con
- $[5, 1, 8, 1, 9, 2]$ là một dãy con
- $[]$ là một dãy con
- $[8, 5]$ **không** là một dãy con
- $[10]$ **không** là một dãy con

Dãy con tăng dài nhất

- Cho một mảng n số nguyên $a[0], a[1], \ldots, a[n-1]$, hãy tìm độ dài của dãy con tăng dài nhất?
- Một dãy con tăng của a là một dãy con của a sao cho các phần tử là tăng chặt từ trái sang phải
- $[5, 8, 9]$ và $[1, 8, 9]$ là hai dãy con tăng của $a = [5, 1, 8, 1, 9, 2]$
- Làm thế nào để tính độ dài dãy con tăng dài nhất?
- Có 2^n dãy con, phương pháp đơn giản nhất là duyệt qua toàn bộ các dãy này
- Thuật toán cho độ phức tạp $O(n \times 2^n)$, chỉ có thể chạy nhanh được ra kết quả với $n \leq 23$
- Phương pháp Qui hoạch động thì sao?

Dãy con tăng dài nhất

- Gọi $\mathrm{lis}(i)$ là độ dài dãy con tăng dài nhất của mảng $a[0], \ldots, a[i]$
- Bước cơ sở: $\mathrm{lis}(0) = 1$
- Bước chuyển để qui cho $\mathrm{lis}(i)$?

Dãy con tăng dài nhất

- Gọi $\text{lis}(i)$ là độ dài dãy con tăng dài nhất của mảng $a[0], \ldots, a[i]$
- Bước cơ sở: $\text{lis}(0) = 1$
- Bước chuyển đệ qui cho $\text{lis}(i)$?
- Nếu đặt hàm mục tiêu như vậy sẽ gặp phải vấn đề giống như bài toán tổng lớn nhất trong mảng ở trên, hãy thay đổi một chút hàm mục tiêu

Dãy con tăng dài nhất

- Gọi $\text{lis}(i)$ là độ dài dãy con tăng dài nhất của mảng $a[0], \ldots, a[i]$, *mà kết thúc tại i*
- Bước cơ sở: không cần thiết
- $\text{lis}(i) = \max(1, \max_{j \text{ s.t. } a[j]<a[i]}\{1 + \text{lis}(j)\})$

Dãy con tăng dài nhất

```
int a[1000];
int mem[1000];
memset(mem, -1, sizeof(mem));

int lis(int i) {
    if (mem[i] != -1) {
        return mem[i];
    }

    int res = 1;
    for (int j = 0; j < i; j++) {
        if (a[j] < a[i]) {
            res = max(res, 1 + lis(j));
        }
    }

    mem[i] = res;
    return res;
}
```

Dãy con tăng dài nhất

- Bây giờ độ dài dãy con tăng dài nhất chính là giá trị lớn nhất trong các giá trị $\mathrm{lis}(i)$:

```
int mx = 0;
for (int i = 0; i < n; i++) {
    mx = max(mx, mem[i]);
}

printf("%d\n", mx);
```

Dãy con tăng dài nhất

- Độ phức tạp tính toán?

Dãy con tăng dài nhất

- Độ phức tạp tính toán?

- Có n khả năng input
- Mỗi input được tính trong thời gian $O(n)$, giả thiết mỗi lời gọi đệ qui chỉ mất $O(1)$
- Thời gian tính tổng cộng là $O(n^2)$

- Có thể chạy được đến $n \leq 10\,000$, tốt hơn rất nhiều so với phương pháp duyệt toàn bộ!

Dãy con tăng dài nhất - Truy vết bằng đệ qui

```
void trace(int i) {
    int res = 1;
    for (int j = 0; j < i; j++) {
        if (a[j] < a[i] && mem[i] == 1 + mem[j]) {
            trace(j);
            break;
        }
    }
    printf("%d ", i);

}
```

Dãy con tăng dài nhất - Truy vết bằng vòng lặp

```
int mx = 0, pos = -1;
for (int i = 0; i < n; i++) {
    mx = max(mx, mem[i]);
    if (mx == mem[i]) pos = i;
}

printf("%d\n", mx);
stack<int> s;
for (int i = pos, k = 0; k < mx; ++k) {
    s.push(i);
    for (int j = 0; j < i; ++j){
        if (a[j] < a[i] && mem[j]+1 == mem[i]){
            i = j;
            break;
        }
    }
    while (!s.empty()){
        printf("%d ", s.back());
        s.pop();
    }
}
```

Dãy con tăng dài nhất - Cải tiến $O(n \log n)$

- Gọi `pos[k]` là vị trí có giá trị trong mảng a nhỏ nhất mà có độ dài dãy con tăng là k

  Tức là `pos[k]=i` khi `lis[i]=k` và `a[i]` là nhỏ nhất trong các i thoả mãn `lis[i]=k`
- Khi đó dãy `a[pos[k]]` tăng dần khi k tăng lên
- Nên khi xét đến i, ta sẽ chặt nhị phân để tìm ra k lớn nhất mà `a[pos[k]] < a[i]`
- Khi đó `lis[i]=k+1`

Qui hoạch động trên bitmask

- Mỗi tập con của tập n phần tử được biểu diễn bởi một số nguyên trong khoảng $0, \ldots, 2^n - 1$
- Điều này có thể giúp thực hiện phương pháp qui hoạch động dễ dàng trên các tập con

Traveling salesman problem

Applying the Traveling Salesman Problem to Business Analytics

Published on April 2, 2016

"The history and evolution of solutions to the Traveling Salesman Problem can provide us with some valuable concepts for business analytics and algorithm development"

"Just as the traveling salesman makes his journey, new analytical requirements arise that require a journey into the development of solutions for them. As the data science and business analytics landscape evolves with new solutions, we can learn from the history of these journeys and apply the same concepts to our ongoing development"

Bài toán người du lịch (hay người bán hàng)

Applying the Traveling Salesman Problem to Business Analytics

Published on April 2, 2016

Bài toán người du lịch là bài toán NP-khó kinh điển nhưng có rất nhiều ứng dụng trong thực tế, đặc biệt ngày nay với ứng dụng của khoa học dữ liệu và phân tích tài chính.

Mỗi khi một hành trình của người bán hàng kết thúc, các dữ liệu sẽ được phân tích bởi các thuật toán trong khoa học dữ liệu, áp dụng vào ngành phân tích tài chính, từ đó có thể 'hoc máy' các kết quả lịch sử để áp dụng vào kế hoạch phát triển tiếp theo.

Bài toán người du lịch

- Cho một đồ thi n đỉnh và giá trị trọng số $c_{i,j}$ trên mỗi cặp đỉnh i, j. Hãy tìm một chu trình đi qua tất cả các đỉnh của đồ thị, mỗi đỉnh đúng một lần sao cho tổng các trọng số trên chu trình đó là nhỏ nhất
- Đây là bài toán NP-khó, vì vậy không tồn tại thuật toán tất định thời gian đa thức nào hiện biết để giải bài toán này
- Thuật toán duyệt toàn bộ đơn giản duyệt qua toàn bộ các hoán vị các đỉnh cho độ phức tạp là $O(n!)$, nhưng chỉ có thể chạy được đến $n \leq 11$
- Liệu có thể làm tốt hơn với phương pháp qui hoạch động?

Bài toán người du lịch

- Không mất tính tổng quát giả sử chu trình bắt đầu và kết thúc tại đỉnh 0
- Gọi $\text{tsp}(i, S)$ cách sử dụng ít chi phí nhất để đi qua toàn bộ các đỉnh và quay trở lại đỉnh 0, nếu như hiện tại hành trình đang ở tại đỉnh i và người du lịch đã thăm tất cả các đỉnh trong tập S

- Bước cơ sở: $\text{tsp}(i, \text{tập mọi đỉnh}) = c_{i,0}$
- Bước chuyển đệ qui: $\text{tsp}(i, S) = \min_{j \notin S} \{ c_{i,j} + \text{tsp}(j, S \cup \{j\}) \}$

Biểu diễn tập hợp

- Cho một số lượng nhỏ ($n \leq 30$) phần tử
- Gán nhãn bởi các số nguyên $0, 1, \ldots, n-1$
- Biểu diễn tập hợp các phần tử này bởi một biến nguyên 32-bit
- Phần thử thứ i trong tập được biểu diễn bởi số nguyên x nếu bit thứ i của x là 1
- Ví dụ:
  - Cho tập hợp $\{0, 3, 4\}$
  - `int x = (1<<0) | (1<<3) | (1<<4);`

Biểu diễn tập hợp

- Tập rỗng:

```
0
```

- Tập có một phần tử:

```
1<<i
```

- Tập vũ trụ (nghĩa là tất cả các phần tử):

```
(1<<n)-1
```

- Hợp hai tập:

```
x|y
```

- Giao hai tập:

```
x&y
```

- Phần bù một tập:

```
~x & ((1<<n)-1)
```

Biểu diễn tập hợp

- Kiểm tra một phần tử xuất hiện trong tập hợp:

```
if (x & (1<<i)) {
    // yes
} else {
    // no
}
```

Biểu diễn tập hợp

- Tại sao nên làm như vậy mà không dùng set<int>?
- Biểu diễn đỡ tốn khá nhiều bộ nhớ (nén 32,64,128 lần)
- Tất cả các tập con của tập n phần tử này có thể biểu diễn bởi các số nguyên trong khoảng $0 \ldots 2^n - 1$
- Dễ dàng lặp qua tất cả các tập con
- Dễ dàng sử dụng một tập hợp như một chỉ số của một mảng

Bài toán người du lịch

```
const int N = 20;
const int INF = 100000000;
int c[N][N];
int mem[N][1<<N];
memset(mem, -1, sizeof(mem));

int tsp(int i, int S) {
    if (S == ((1 << N) - 1))  return c[i][0];

    if (mem[i][S] != -1) {
        return mem[i][S];
    }
    int res = INF;
    for (int j = 0; j < N; j++) {
        if (S & (1 << j))
            continue;
        res = min(res, c[i][j] + tsp(j, S | (1 << j)));
    }
    mem[i][S] = res;
    return res;
}
```

Bài toán người du lịch

- Như vậy lời giải tối ưu có thể được đưa ra như sau:
- `printf("%d\n", tsp(0, 1<<0));`

Bài toán người du lịch

- Độ phức tạp tính toán?

Bài toán người du lịch

- Độ phức tạp tính toán?

- Có $n \times 2^n$ khả năng input
- Mỗi input được tính trong thời gian $O(n)$, giả thiết mỗi lời gọi đệ qui chỉ mất $O(1)$
- Thời gian tính tổng cộng là $O(n^2 \times 2^n)$
- Như vậy có thể tính nhanh được với n lên đến 20

Bài toán người du lịch

- Độ phức tạp tính toán?

- Có $n \times 2^n$ khả năng input
- Mỗi input được tính trong thời gian $O(n)$, giả thiết mỗi lời gọi đệ qui chỉ mất $O(1)$
- Thời gian tính tổng cộng là $O(n^2 \times 2^n)$
- Như vậy có thể tính nhanh được với n lên đến 20

- Làm thế nào để đưa ra được chính xác hành trình của người du lịch?

Bài toán người du lịch - Truy vết bằng đệ qui

```
void trace_tsp(int i, int S) {
    printf("%d ", i);
    if (S == ((1 << N) - 1))  return;

    int res = mem[i][S];
    for (int j = 0; j < N; j++) {
        if (S & (1 << j))
            continue;

        if (res == c[i][j] + mem[j][S | (1 << j)]){
            trace_tsp(j, S | (j << j));
            break;
        }
    }
}
```

Bài toán người du lịch - Truy vết bằng vòng lặp

```
int answer = tsp(0, 1);
printf("%d\n", answer);
stack<int> s;
s.push(0);
for (int i = 0, S = 1, k = 0; k < n-1; ++k) {
   for (int j = 0; j < n; ++j){
     if ((S & (1 << j))
        && (mem[i][S] == c[i][j] + mem[j][S | (1 << j)])){
            s.push(j);
            i = j;
            S = S | (1 << j);
     }
   }
}
while (!s.empty()) {
    printf("%d ", s.back());
    s.pop();
}
```