ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

Qui Hoạch Động

THUẬT TOÁN ỨNG DỤNG

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 'multistage decision 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 'multistage' 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 qui hoạch tuyến tính), tuy nhiên từ dịch ra 'Qui Hoạch Động' nghe hay và thuận hơn từ 'Lập Trình Động'.

Các mô hình giải bài căn bản

Các phương pháp căn bản xây dựng lời giải cho từng dạng bài toán

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

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

1. Sơ đồ Qui hoạch động
2. Tính số Fibonacci
3. Đoạn con có tổng lớn nhất
4. Đổi tiền
5. Dãy con tăng dài nhất
6. Dãy con chung dài nhất
7. Qui hoạch động trên bitmask

1. Sơ đồ Qui hoạch động
   - Qui hoạch động là gì?
   - Công thức Qui hoạch động
   - Cài đặt Top-Down với Đệ qui có nhớ
2. Tính số Fibonacci
3. Đoạn con có tổng lớn nhất
4. Đổi tiền
5. Dãy con tăng dài nhất
6. Dãy con chung dài nhất
7. Qui hoạch động trên bitmask

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:
   Đơn giản là 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

Cài đặt Top-Down với Đệ qui có nhớ

```
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
- 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
- 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

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

1. Tìm công thức truy hồi:

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

Tính số Fibonacci

2. Cài đặt công thức qui hoạch động

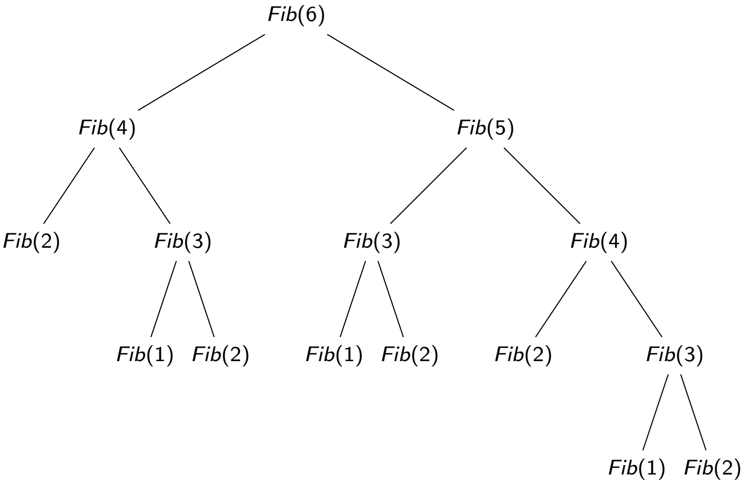
```
int Fib(int n) {
    if (n <= 2)
        return 1;

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

    return res;
}
```

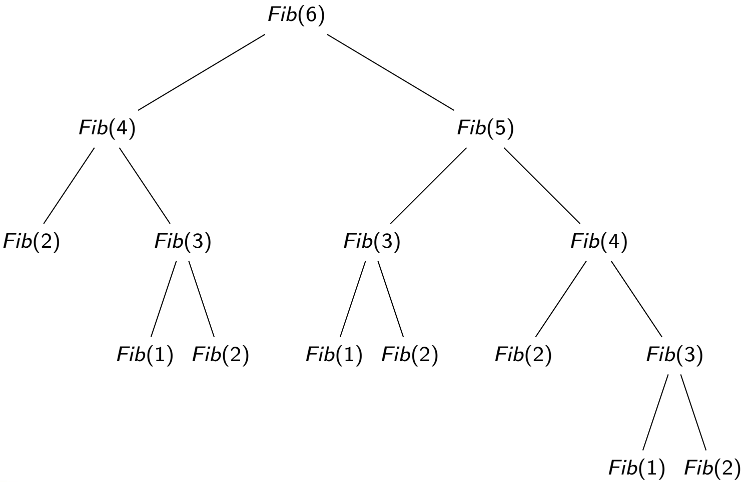
Tính số Fibonacci

- Độ phức tạp là bao nhiêu?



Tính số Fibonacci

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



Tính số 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 = Fib(n - 2) + Fib(n - 1);

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

Dãy Fibonacci

```
int iMem[1001];
for (int i = 1; i <= 1000; i++)
    iMem[i] = -1;

int Fib(int n) {
    if (n <= 2)
        return 1;
    if (iMem[n] != -1)
        return iMem[n];

    int res = Fib(n - 2) + Fib(n - 1);
    iMem[n] = res;
    return res;
}
```

- Bây giờ độ phức tạp là bao nhiêu?

Tính số Fibonacci: Độ phức tạp

- Ta có n khả năng đầu vào input cho hàm đệ qui: $1, 2, \ldots, n$.
- 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 $\mathcal{O}(n \times f)$, 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 $\mathcal{O}(1)$
- 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 $\mathcal{O}(n)$

1. Sơ đồ Qui hoạch động
2. Tính số Fibonacci
3. Đoạn con có tổng lớn nhất
   - Bài toán
   - Công thức Qui hoạch động
   - Cài đặt
   - Độ phức tạp
   - Truy vết
4. Đổi tiền
5. Dãy con tăng dài nhất
6. Dãy con chung dài nhất
7. Qui hoạch động trên bitmask

Đoạn con có tổng lớn nhất

- Cho một dãy số nguyên $A[1], A[2], \ldots, A[n]$, hãy tìm một đoạn trong dãy có trọng số lớn nhất, nghĩa là tổng các số trong đoạn là lớn nhất

-16	7	-3	0	-1	5	-4

Đoạn con có tổng lớn nhất

- Cho một dãy số nguyên $A[1], A[2], \ldots, A[n]$, hãy tìm một đoạn trong dãy có trọng số lớn nhất, nghĩa là tổng các số trong đoạn là lớn nhất

-16	7	-3	0	-1	5	-4

- Tổng của đoạn có trọng số lớn nhất trong dãy là 8

Đoạn con có tổng lớn nhất

- Cho một dãy số nguyên $A[1], A[2], \ldots, A[n]$, hãy tìm một đoạn trong dãy có trọng số lớn nhất, nghĩa là tổng các số trong đoạn là lớn nhất

-16	7	-3	0	-1	5	-4

- Tổng của đoạn có trọng số lớn nhất trong dãy là 8
- Nhớ lại:
  - Phương pháp Chia để trị cho độ phức tạp $\mathcal{O}(n \log n)$
  - Liệu có thể làm tốt hơn với phương pháp Qui hoạch động?

Đoạn con có tổng lớn nhất: Công thức sai

- Bước đầu tiên là đi tìm công thức Qui hoạch động
- Gọi $\mathrm{MaxSum}(i)$ là trọng số của đoạn có trọng số lớn nhất giới hạn trong đoạn $1, \ldots, i$
- Bước cơ sở: $\mathrm{MaxSum}(1) = \max(0, A[1])$
- Bước chuyển qui nạp: $\mathrm{MaxSum}(i)$ có liên hệ gì với $\mathrm{MaxSum}(i-1)$?
- 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?
- Câu trả lời không hoàn toàn hiển nhiên ...

Đoạn con có tổng lớn nhất: Công thức

Hãy thay đổi hàm mục tiêu:

- Gọi $\mathrm{MaxSum}(i)$ là trọng số đoạn có trọng số lớn nhất giới hạn bởi $1, \ldots, i$, *mà phải kết thúc tại i*
- Bước cơ sở: $\mathrm{MaxSum}(1) = A[1]$
- Bước chuyển qui nạp:
  $\mathrm{MaxSum}(i) = \max(A[i], A[i] + \mathrm{MaxSum}(i-1))$
- Vậy kết quả cuối cùng chính là: $\max_{1 \leq i \leq n} \{ \mathrm{MaxSum}(i) \}$

Đoạn con có tổng lớn nhất: Cài đặt

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

```
int A[1001];

int MaxSum(int i) {
    if (i == 1)
        return A[i];

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

Đoạn con có tổng lớn nhất: Cài đặt

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

```
int A[1001];

int MaxSum(int i) {
    if (i == 1)
        return A[i];

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

- Không cần sử dụng mảng nhớ `Memory`, vì sao?

Đoạn con có tổng lớn nhất: Cài đặt

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

```
int A[1001];

int MaxSum(int i) {
    if (i == 1)
        return A[i];

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

- Không cần sử dụng mảng nhớ `Memory`, vì sao?
- Nhưng vẫn cần mảng nhớ `Memory` phục vụ cho bước truy vết!

Đoạn con có tổng lớn nhất: Cài đặt

```
int A[1001];
int iMem[1001];
bool bMark[1001];
memset(bMark, 0, sizeof(bMark));

int MaxSum(int i) {
    if (i == 1)
        return A[i];
    if (bMark[i])
        return iMem[i];

    int res = max(A[i], A[i] + MaxSum(i - 1));
    iMem[i] = res;
    bMark[i] = true;
    return res;
}
```

Đoạn con có tổng lớn nhất: Kết quả

- Thủ tục chính chỉ cần gọi đệ qui một lần cho $\mathrm{MaxSum}(n)$, hàm đệ qui sẽ tính toàn bộ các giá trị của $\mathrm{MaxSum}(i), 1 \leq i \leq n$
- Kết quả bài toán là giá trị lớn nhất trong các giá trị $\mathrm{MaxSum}(i)$ đã được lưu trữ trong $\mathrm{iMem}[i]$ sau quá trình gọi đệ qui

```
int ans = 0;
for (int i = 0; i < n; i++) {
    ans = max(ans, iMem[i]);
}
cout << ans;
```

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

Đoạn con có tổng lớn nhất: Độ phức tạp

- Có n khả năng đầu vào input cho hàm đệ qui
- Mỗi input được tính trong $\mathcal{O}(1)$
- Thời gian tính toán tổng cộng là $\mathcal{O}(n)$

Đoạn con có tổng lớn nhất: Độ phức tạp

- Có n khả năng đầu vào input cho hàm đệ qui
- Mỗi input được tính trong $\mathcal{O}(1)$
- Thời gian tính toán tổng cộng là $\mathcal{O}(n)$
- Làm thế nào để biết chính xác một đoạn nào trong dãy tạo ra giá trị tổng lớn nhất tìm được?

Đoạn con có tổng lớn nhất: Truy vết bằng đệ qui

- Làm giống như hàm đệ qui chính và sử dụng mảng `iMem` đã có để truy vết ngược lại

```
void Trace(int i) {
  if (i != 1 && iMem[i] == A[i] + iMem[i-1]) {
     Trace(i - 1);
  }
  cout << A[i] << " ";
}
```

- Độ phức tạp hàm truy vết?

Đoạn con có tổng lớn nhất: Truy vết bằng đệ qui

- Làm giống như hàm đệ qui chính và sử dụng mảng `iMem` đã có để truy vết ngược lại

```
void Trace(int i) {
  if (i != 1 && iMem[i] == A[i] + iMem[i-1]) {
      Trace(i - 1);
  }
  cout << A[i] << " ";
}
```

- Độ phức tạp hàm truy vết? $\mathcal{O}(n)$

Đoạn con có tổng lớn nhất: Truy vết bằng vòng lặp

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

cout << ans << endl;
int first = pos, last = pos, sum = A[first];
while (sum != res){
    --first;
    sum += A[first];
}
cout << first << " " << last;
```

1 Sơ đồ Qui hoạch động

2 Tính số Fibonacci

3 Đoạn con có tổng lớn nhất

4 Đổi tiền
- Bài toán
- Công thức Qui hoạch động
- Cài đặt
- Độ phức tạp
- Truy vết

5 Dãy con tăng dài nhất

6 Dãy con chung dài nhất

7 Qui hoạch động trên bitmask

Đổi tiền

- Cho trước một tập các đồng tiền mệnh giá $D_1, D_2, \ldots, D_n$, và một mệnh giá x. Hãy tìm số lượng ít nhất các đồng tiền để đổi cho mệnh giá x?
- Giống bài toán cái túi?
- Tồn tại thuật toán tham lam cho bài Đổi tiền này?

Đổi tiền

- Cho trước một tập các đồng tiền mệnh giá $D_1, D_2, \ldots, D_n$, và một mệnh giá x. Hãy tìm số lượng ít nhất các đồng tiền để đổi cho mệnh giá x?
- Giống bài toán cái túi?
- Tồn tại thuật toán tham lam cho bài Đổi tiền này?
- Bài toán cái túi đã học trong môn `Toán rời rạc` được giải bằng thuật toán duyệt nhánh cận. Còn 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...
- Hãy thử sử dụng phương pháp Qui hoạch động !
- Cuối cùng đưa ra nhận xét giữa các phương pháp khác nhau tiếp cận giải bài toán này

Đổi tiền: Công thức Qui hoạch động

Bước đầu tiên: xây dựng công thức Qui hoạch động

- Gọi $\mathrm{MinCoin}(i, x)$ là số lượng tiền ít nhất cần để đổi mệnh giá x nếu chỉ được phép sử dụng các đồng tiền mệnh giá $D_1, \ldots, D_i$
- Các bước cơ sở:
  - $\mathrm{MinCoin}(i, x) = \infty$ nếu $x < 0$
  - $\mathrm{MinCoin}(i, 0) = 0$
  - $\mathrm{MinCoin}(0, x) = \infty$
- Bước chuyển qui nạp:

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

Đổi tiền: Cài đặt

```
int INF = 100000;
int D[11];

int MinCoin(int i, int x) {
    if (x < 0) return INF;
    if (x == 0) return 0;
    if (i == 0) return INF;

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

    return res;
}
```

Đổi tiền: Cài đặt

```
int INF = 100000;
int D[11];
int iMem[11][10001];
memset(iMem, -1, sizeof(iMem));

int MinCoin(int i, int x) {
    if (x < 0) return INF;
    if (x == 0) return 0;
    if (i == 0) return INF;

    if (iMem[i][x] != -1) return iMem[i][x];
    int res = INF;
    res = min(res, 1 + MinCoin(i, x - D[i]));
    res = min(res, MinCoin(i - 1, x));
    iMem[i][x] = res;
    return res;
}
```

Đổi tiền: Độ phức tạp

- Độ phức tạp?

Đổi tiền: Độ phức tạp

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

Đổi tiền: Độ phức tạp

- Độ phức tạp?
- Số lượng khả năng đầu vào input là $n \times x$
- Mỗi input được xử lý trong $\mathcal{O}(1)$, giả thiết mỗi lời gọi đệ qui thực hiện trong thời gian hằng số
- Thời gian tính toán tổng cộng là $\mathcal{O}(n \times x)$
- 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 == 0) return;

    int res = INF;
    if (iMem[i][x] == 1 + iMem[i][x - D[i]]){
        cout << D[i] << " ";
        Trace(i, x - D[i]);
    } else {
        Trace(i-1, x);
    }
}
```

- Gọi `Trace(n,x);`
- Độ phức tạp hàm truy vết?

Đổi tiền: Truy vết bằng đệ qui

```
void Trace(int i, int x) {
    if (x < 0) return;
    if (x == 0) return;
    if (i == 0) return;

    int res = INF;
    if (iMem[i][x] == 1 + iMem[i][x - D[i]]){
        cout << D[i] << " ";
        Trace(i, x - D[i]);
    } else {
        Trace(i-1, x);
    }
}
```

- Gọi `Trace(n,x);`
- Độ phức tạp hàm truy vết? $\mathcal{O}(\max(n, x))$

Đổi tiền: Truy vết bằng vòng lặp

```
int ans = iMem[n][x];
cout << ans << endl;
for (int i = n, k = 0; k < ans; ++k) {
    if (iMem[i][x] == 1 + iMem[i][x-D[i]]){
        cout << D[i] << " ";
        x -= D[i];
    } else {
        --i;
    }
}
```

1. Sơ đồ Qui hoạch động
2. Tính số Fibonacci
3. Đoạn con có tổng lớn nhất
4. Đổi tiền
5. Dãy con tăng dài nhất
   - Bài toán
   - Công thức Qui hoạch động
   - Cài đặt
   - Độ phức tạp
   - Truy vết
6. Dãy con chung dài nhất
7. Qui hoạch động trên bitmask

Dãy con tăng dài nhất

- Cho một dãy n số nguyên $A[1], A[2], \ldots, A[n]$, hãy tìm độ dài của dãy con tăng dài nhất?

- Định nghĩa: Nếu xoá đi 0 phần tử hoặc một số phần tử của dãy A thì sẽ thu được một dãy con của A

- Ví dụ: $a = [2, 0, 6, 1, 2, 9]$

- $[2, 6, 9]$ là một dãy con
- $[2, 2]$ là một dãy con
- $[2, 0, 6, 1, 2, 9]$ là một dãy con
- $[]$ là một dãy con
- $[9, 0]$ **không** là một dãy con
- $[7]$ **không** là một dãy con

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
- $[2, 6, 9]$ và $[1, 2, 9]$ là hai dãy con tăng của $A = [2, 0, 6, 1, 2, 9]$
- 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 $\mathcal{O}(n \times 2^n)$, chỉ có thể chạy nhanh được ra kết quả với $n \leq 23$
- Hãy thử phương pháp Qui hoạch động !

Dãy con tăng dài nhất: Công thức Qui hoạch động

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

Dãy con tăng dài nhất: Công thức Qui hoạch động

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

- Nếu đặt hàm mục tiêu như vậy sẽ gặp phải vấn đề giống như bài toán dãy con có tổng lớn nhất ở trên, hãy thay đổi một chút hàm mục tiêu

Dãy con tăng dài nhất: Công thức Qui hoạch động

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

Dãy con tăng dài nhất: Cài đặt

```
int A[1001];
int iMem[1001];
memset(iMem, -1, sizeof(iMem));

int LIS(int i) {
    if (iMem[i] != -1)
        return iMem[i];

    int res = 1;
    for (int j = 1; j < i; ++j) {
        if (A[j] < A[i]) {
            res = max(res, 1 + LIS(j));
        }
    }
    iMem[i] = res;
    return res;
}
```

Dãy con tăng dài nhất: Cài đặt

- Độ dài dãy con tăng dài nhất chính là giá trị lớn nhất trong các giá trị $\mathrm{LIS}(i)$:

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

cout << ans;
```

Dãy con tăng dài nhất: Độ phức tạp tính toán

- Có n khả năng cho đầu vào input
- Mỗi input được tính trong thời gian $\mathcal{O}(n)$
- Thời gian tính tổng cộng là $\mathcal{O}(n^2)$
- Có thể chạy được đến $n \leq 10\,000$, tốt hơn rất nhiều so với phương pháp duyệt toàn bộ!
- Áp dụng cấu trúc cây phân đoạn vào phương pháp trên sẽ cải tiến độ phức tạp thành $\mathcal{O}(n \log n)$
- Phương pháp cải tiến khác là đặt công thức Qui hoạch động mới kết hợp với phương pháp chặt nhị phân cũng cho độ phức tạp $\mathcal{O}(n \log n)$
- Truy vết ?

Dãy con tăng dài nhất: Truy vết bằng đệ qui

```
void Trace(int i) {
      int res = 1;
      for (int j = 1; j < i; j++) {
        if (A[j] < A[i] && iMem[i] == 1 + iMem[j]) {
          Trace(j);
          break;
        }
      }
      cout << i << " ";
}
```

- Gọi `Trace(pos);`
- Độ phức tạp hàm truy vết?

Dãy con tăng dài nhất: Truy vết bằng đệ qui

```
void Trace(int i) {
      int res = 1;
      for (int j = 1; j < i; j++) {
        if (A[j] < A[i] && iMem[i] == 1 + iMem[j]) {
          Trace(j);
          break;
        }
      }
      cout << i << " ";
}
```

- Gọi `Trace(pos);`
- Độ phức tạp hàm truy vết? vẫn là $\mathcal{O}(n^2)$
- Có thể cải tiến thành $\mathcal{O}(n)$ bằng cách sử dụng một mảng nhớ lưu tại mỗi vị trí i vị trí j làm tạo ra giá trị $\max(LIS(i))$ để giảm bớt vòng lặp `for` trong hàm truy vết này

Dãy con tăng dài nhất: Truy vết bằng đệ qui cải tiến

```
void Trace(int i) {
      int res = 1;
      for (int j = i-1; j >= 1; j++) {
        if (A[j] < A[i] && iMem[i] == 1 + iMem[j]) {
          Trace(j);
          break;
        }
      }
      cout << i << " ";
}
```

- Gọi `Trace(pos);`
- Độ phức tạp hàm truy vết?

Dãy con tăng dài nhất: Truy vết bằng đệ qui cải tiến

```
void Trace(int i) {
      int res = 1;
      for (int j = i-1; j >= 1; j++) {
         if (A[j] < A[i] && iMem[i] == 1 + iMem[j]) {
            Trace(j);
            break;
         }
      }
      cout << i << " ";
}
```

- Gọi `Trace(pos);`
- Độ phức tạp hàm truy vết? $\mathcal{O}(n)$

Dãy con tăng dài nhất: Truy vết bằng vòng lặp

```
stack<int> S;
for (int i = pos, k = 0; k < ans; ++k) {
    S.push(i);
    for (int j = 1; j < i; ++j){
        if (A[j] < A[i] && iMem[j]+1 == iMem[i]) {
            i = j;
            break;
        }
    }
    while (!S.empty()){
        cout << S.back() << " ";
        S.pop();
    }
}
```

Dãy con chung dài nhất

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

- $X =$"a<u>bcb</u>"
- $Y =$"<u>b</u>d<u>c</u>a<u>b</u>"

- Dãy con chung dài nhất của X và Y, "bcb", có độ dài 3

Dãy con chung dài nhất: Công thức Qui hoạch động

- Gọi $\mathrm{LCS}(i, j)$ là độ dài dãy con chung dài nhất của $X[1], \ldots, X[i]$ và $Y[1], \ldots, Y[j]$
- Bước cơ sở:
  - $\mathrm{LCS}(0, j) = 0$
  - Bước cơ sở: $\mathrm{LCS}(i, 0) = 0$
- Bước chuyển qui nạp:

$$\mathrm{LCS}(i, j) = \max \begin{cases} \mathrm{LCS}(i, j-1) \\ \mathrm{LCS}(i-1, j) \\ 1 + \mathrm{LCS}(i-1, j-1) & \text{nếu } X[i] = Y[j] \end{cases}$$

Dãy con chung dài nhất: Cài đặt

```
string X = "abcb",
       Y = "bdcab";
int iMem[1001][1001];
memset(iMem, -1, sizeof(iMem));

int LCS(int i, int j) {
    if (i == 0 || j == 0)  return 0;
    if (iMem[i][j] != -1)  return iMem[i][j];

    int res = 0;
    res = max(res, LCS(i, j - 1));
    res = max(res, LCS(i - 1, j));
    if (X[i] == Y[j]) {
        res = max(res, 1 + LCS(i - 1, j - 1));
    }
    iMem[i][j] = res;
    return res;
}
```

Dãy con chung dài nhất: Ví dụ

iMem	j	0	1	2	3	4	5
i		Y[j]	**b**	**d**	**c**	**a**	**b**
0	X[i]	0	0	0	0	0	0
1	**a**	0 $\searrow$	0	0	0	1	1
2	**b**	0	1 $\rightarrow$	1 $\searrow$	1	1	2
3	**c**	0	1	1	2 $\rightarrow$	2 $\searrow$	2
4	**b**	0	1	1	2	2	3

$$\mathrm{LCS}(i,j) = \max \begin{cases} \mathrm{LCS}(i,j-1) \\ \mathrm{LCS}(i-1,j) \\ 1 + \mathrm{LCS}(i-1,j-1) & \text{nếu } X[i] = Y[j] \end{cases}$$

Dãy con chung dài nhất: Độ phức tạp tính toán

- Có n khả năng cho đầu vào input
- Mỗi input được tính trong thời gian $\mathcal{O}(1)$
- Thời gian tính tổng cộng là $\mathcal{O}(n \times m)$

Dãy con chung dài nhất: Độ phức tạp tính toán

- Có n khả năng cho đầu vào input
- Mỗi input được tính trong thời gian $\mathcal{O}(1)$
- Thời gian tính tổng cộng là $\mathcal{O}(n \times m)$
- Làm thế nào để biết chính xác những phần tử nào thuộc dãy con chung dài nhất?

Dãy con chung dài nhất: Truy vết bằng đệ qui

```
void Trace(int i, int j) {
    if (i == 0 || j == 0)  return;

    if (iMem[i][j] == iMem[i-1][j]) {
        Trace(i-1, j);
        return;
    }
    if (iMem[i][j] == iMem[i][j-1]) {
        Trace(i, j-1);
        return;
    }
    if (X[i] == Y[j] && iMem[i][j] == 1 + iMem[i-1][j-1]) {
        Trace(i-1, j-1);
        cout << A[i] << " ";
        return;
    }
}
```

- Độ phức tạp hàm truy vết?

Dãy con chung dài nhất: Truy vết bằng đệ qui

```
void Trace(int i, int j) {
    if (i == 0 || j == 0)  return;

    if (iMem[i][j] == iMem[i-1][j]) {
        Trace(i-1, j);
        return;
    }
    if (iMem[i][j] == iMem[i][j-1]) {
        Trace(i, j-1);
        return;
    }
    if (X[i] == Y[j] && iMem[i][j] == 1 + iMem[i-1][j-1]) {
        Trace(i-1, j-1);
        cout << A[i] << " ";
        return;
    }
}
```

- Độ phức tạp hàm truy vết? $\mathcal{O}(n+m)$

Dãy con chung dài nhất - Truy vết bằng vòng lặp

```
int ans = iMem[n][m];
cout << ans << endl;
stack<int> S;
for (int i = n, j = m, k = 0; k < ans; ++k) {
    if (X[i] == Y[j] && iMem[i][j] == 1 + iMem[i-1][j-1]){
        S.push(X[i]);
        --i; --j; continue;
    }
    if (iMem[i][j] == iMem[i-1][j]){
        --i; continue;
    }
    if (iMem[i][j] == iMem[i][j-1]){
        --j; continue;
    }
}
while (!S.empty()) {
    cout << S.back() << " ";
    S.pop();
}
```

7. Qui hoạch động trên bitmask
   - Bài toán người du lịch
   - Công thức Qui hoạch động
   - Cài đặt
   - Độ phức tạp
   - Truy vết

Qui hoạch động trên bitmask

- Có còn nhớ biểu diễn bitmask cho các tập con?
- 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

Bài toán người du lịch

"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 'học 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 $\{0,1,\ldots,n-1\}$ 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 $\mathcal{O}(n!)$, 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: Công thức Qui hoạch động

- Không mất tính tổng quát giả sử chu trình bắt đầu và kết thúc tại đỉnh 0
- Gọi $\mathrm{TSP}(i, S)$ 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: $\mathrm{TSP}(i, \text{tập mọi đỉnh}) = C_{i,0}$
- Bước chuyển qui nạp: $\mathrm{TSP}(i, S) = \min_{j \notin S} \{\ C_{i,j} + \mathrm{tsp}(j, S \cup \{j\})\ \}$

Bài toán người du lịch: Cài đặt

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

int TSP(int i, int S) {
    if (S == ((1 << N) - 1))  return C[i][0];
    if (iMem[i][S] != -1)     return iMem[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)));
    }
    iMem[i][S] = res;
    return res;
}
```

Bài toán người du lịch: Cài đặt

- Kết quả tối ưu có thể được đưa ra như sau:

```
cout << TSP(0, 1<<0);
```

Bài toán người du lịch: Độ phức tạp tính toán

- Có $n \times 2^n$ khả năng cho đầu vào input
- Mỗi input được tính trong thời gian $\mathcal{O}(n)$
- Thời gian tính tổng cộng là $\mathcal{O}(n^2 \times 2^n)$
- 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 cho đầu vào input
- Mỗi input được tính trong thời gian $\mathcal{O}(n)$
- Thời gian tính tổng cộng là $\mathcal{O}(n^2 \times 2^n)$
- 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(int i, int S) {
    cout << i << " ";
    if (S == ((1 << N) - 1))  return;

    int res = iMem[i][S];
    for (int j = 0; j < N; j++) {
      if (S & (1 << j))
        continue;
      if (res == C[i][j] + iMem[j][S | (1 << j)]) {
        Trace(j, S | (1 << j));
        break;
      }
    }
}
```

- Gọi `TSP(0, 1«0);`
- Độ phức tạp hàm truy vết?

Bài toán người du lịch - Truy vết bằng đệ qui

```
void Trace(int i, int S) {
    cout << i << " ";
    if (S == ((1 << N) - 1))  return;

    int res = iMem[i][S];
    for (int j = 0; j < N; j++) {
      if (S & (1 << j))
        continue;
      if (res == C[i][j] + iMem[j][S | (1 << j)]) {
        Trace(j, S | (1 << j));
        break;
      }
    }
}
```

- Gọi `TSP(0, 1«0);`
- Độ phức tạp hàm truy vết? $\mathcal{O}(n^2)$

Bài toán người du lịch: Truy vết bằng vòng lặp

```
int ans = iMem[0][1];
cout << ans << endl;
stack<int> Stack;
Stack.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)) &&
      (iMem[i][S] == C[i][j] + iMem[j][S | (1 << j)])) {
        Stack.push(j);
        i = j;
        S = S | (1 << j);
    }
  }
}
while (!Stack.empty()) {
    cout << Stack.back() << " ";
    Stack.pop();
}
```

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Thank you for your attentions!

soict.hust.edu.vn/ fb.com/groups/soict