Chương 1

ĐỘ PHỨC TẠP
CỦA THUẬT TOÁN

Nội dung

- Độ phức tạp của thuật toán
- Ước lượng độ phức tạp của thuật toán

ĐỘ PHỨC TẠP CỦA THUẬT TOÁN

Thời gian thực hiện thuật toán

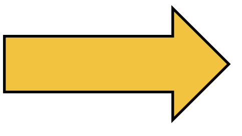
- Phân tích thuật toán: Phân tích thuật toán là xác định lượng tài nguyên cần thiết để thực thi thuật toán:
  - Thời gian thực hiện thuật toán
  - Bộ nhớ cần thực hiện thuật toán
- Tiêu chí thường được dùng để đánh giá thuật toán là thời gian thực hiện thuật toán.

Thời gian thực hiện thuật toán

- Mục tiêu của phân tích thuật toán
  - So sánh để chọn ra thuật toán nào chạy nhanh nhất
  - Tìm những yếu điểm của thuật toán để Cải tiến thuật toán tốt hơn
- 2 cách “đo” thời gian thực hiện của thuật toán
  - Thời gian thực hiện thực tế
  - Thời gian thực hiện lý thuyết (Phân tích thuật toán)

Thời gian thực hiện thuật toán

- Thời gian thực hiện thực tế: Dựa trên thực tế khi chạy các thuật toán được tình bằng (mili second, second, minute, hour, day)



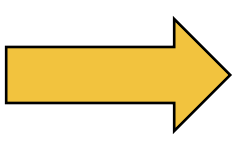
Kết luận: Thuật toán nào nhanh, thuật toán nào chậm

Thời gian thực hiện thuật toán

- Thời gian thực hiện thực tế phụ thuộc vào nhiều yếu tố:
  - Dữ liệu vào:
    - Kích thước dữ liệu
    - Đặc điểm của dữ liệu
  - Tốc độ của máy tính
  - Ngôn ngữ lập trình
  - Chương trình dịch cho ngôn ngữ lập trình
  - Hệ điều hành để thực hiện chương trình

Thời gian thực hiện thuật toán

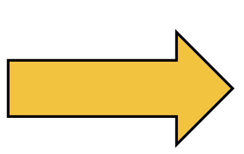
- Thời gian thực hiện thực tế: Dựa trên thực tế khi chạy các thuật toán được viết trên:
  - Cùng ngôn ngữ lập trình, cùng trình biên dịch
  - Cùng hệ thống máy tính
  - Cùng bộ dữ liệu vào chuẩn



Kết luận: Thuật toán nào nhanh, thuật toán nào chậm

Thời gian thực hiện thuật toán

- Thời gian thực hiện lý thuyết: Dựa vào
  - Số phép toán cơ bản trong thuật toán sẽ được thực hiện bao nhiêu lần
  - Kích thước dữ liệu vào



Kết luận

+ Thuật toán nào nhanh, thuật toán nào chậm

+ Tìm ra những nơi cần cải tiến thuật toán

Thời gian thực hiện thuật toán

- Phép toán cơ bản: Một phép toán được gọi là cơ bản nếu thời gian thực hiện của nó bị chặn trên bởi một hằng số (chỉ phụ thuộc cách cài đặt được sử dụng – ngôn ngữ lập trình, máy tính, …).

- Ví dụ:
  - +, -, *, /
  - Các phép so sánh: >, <, >=, <=, ==, !=
  - Phép gán: =, +=, …
  - Đọc file, ghi file
  - cout, cin, printf, scanf
  - …

Thời gian thực hiện thuật toán

- Định nghĩa [Thời gian thực hiện thuật toán]:
  Gọi T(n) là số phép toán cơ bản khi thực hiện thuật toán với kích thước dữ liệu vào n. T(n) được gọi là thời gian thực hiện thuật toán.

- Chú ý: Thuật toán có nhiều loại phép toán cơ bản nên chúng ta có thể thực hiện đánh theo một trong hay cách:
  - Đánh giá thời gian chạy trên từng loại phép toán
  - Tổng hợp các phép toán và gán trọng số cho từng phép toán
  - Xem các phép toán là như nhau

Thời gian thực hiện thuật toán

- Ví dụ: Tìm thời gian thực hiện của thuật toán

```
// Thuật toán tính tổng S=a[0]+a[1]+…+a[n-1]
{1} s = 0;
{2} for (i=0; i<n; i++)
{3}   s = s + a[i];
```

Thời gian thực hiện thuật toán

- Ví dụ: Tìm thời gian thực hiện của thuật toán

```
// Thuật toán tìm max
{1} max = a[0];
{2} for (i=1; i<n; i++)
{3}    if (max < a[i])
{4}            max=a[i];
```

- Nhận xét: Số lần thực hiện của Câu lệnh {4} phụ thuộc vào biểu thức điều kiện trong câu lệnh {3} hay bộ dữ liệu input

Thời gian thực hiện thuật toán

- 3 trường hợp đánh giá thời gian thực hiện thuật toán
  - Trường hợp xấu nhất (worst case): T(n) là thời gian lớn nhất khi thực hiện thuật toán với mọi bộ dữ liệu kích thước n
  - Trường hợp tốt nhất (best case): T(n) là thời gian ít nhất khi thực hiện thuật toán với mọi bộ dữ liệu kích thước n
  - Trường hợp trung bình (average case): Dữ liệu tuân theo 1 phân bố xác suất nào đó. Giả sử P(input) là xác suất dữ liệu input xuất hiện, khi đó thời gian trung bình của thuật toán là

$$T(n) \qquad (input)$$

$input$

Thời gian thực hiện thuật toán

- Ví dụ: Tìm thời gian thực hiện của thuật toán trong trường hợp xấu nhất

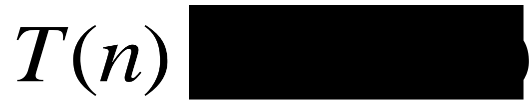
```
// Thuật toán tìm max
{1} max = a[0];
{2} for (i=1; i<n; i++)
{3}    if (max < a[i])
{4}          max=a[i];
```

Độ phức tạp thuật toán

- Nhận xét:
  - Việc đánh giá thời gian thực hiện thuật toán qua hàm T(n) như trên là quá chi tiết. Cho nên việc dùng T(n) để so sánh tính hiệu quả giữa các thuật toán sẽ gặp khó khăn.
  - Để giải quyết khó khăn này Bachmann và Landau giới thiệu khái niệm hàm O (đọc là ô lớn) để xác định độ lớn của hàm T(n)

Độ phức tạp thuật toán

- Định nghĩa [Độ phức tạp thuật toán]:
  - Độ lớn của thời gian thuật toán T(n) được gọi là độ phức tạp thuật toán
  - Giả sử f(n) là hàm xác định dương trên mọi n. Khi đó ta nói độ phức tạp của thuật toán có thời gian thực hiện T(n) là
    - Hàm O (đọc là ô lớn): O(f(n)) nếu tồn tại các hằng số c và n_0 sao cho với mọi $n \geq n_0$ ta có $T(n) \leq c.f(n)$, hàm f(n) được gọi là giới hạn trên của hàm T(n)



Độ phức tạp thuật toán

- Ví dụ: Nếu $T(n)=n^3+3n^2+n+1$ thì $T(n)=O(n^3)$
  - Thật vậy, với mọi $n \geq 1$ ta có:

    $$T(n) = n^3+3n^2+n+1 \leq n^3+3n^3+n^3+n^3=6n^3$$

  - Vậy ta chọn $n_0=1$, $c=6$ và $f(n)=n^3$, ta có: $T(n) \leq c.f(n)$
  - Tóm lại: $T(n)=O(n^3)$

- Nhận xét:
  - Có nhiều hàm f(n) làm chặn trên của T(n)
  - Thông thường người ta chọn f(n) nhỏ nhất và đơn giản nhất có thể

Một số dạng hàm kí hiệu độ phức tạp thuật toán

- Một số hàm f(n) thường dùng để kí hiệu độ phức tạp thuật toán
  - log(n)
  - n
  - n.log(n)
  - $n^{1.25}$, n^2, n^3, n^4,
  - 2^n
  - n!

Các quy tắc của độ phức tạp

- Quy tắc Hằng số: Nếu thuật toán T có độ phức tạp là $T(n)=O(c_1.f(n))$ với c_1 là một hằng số dương thì có thể coi thuật toán T có độ phức tạp là $O(f(n))$
- Chứng minh:

Các quy tắc của độ phức tạp

- Quy tắc Cộng: Nếu thuật toán T gồm 2 phần liên tiếp T_1 và T_2 và
  - Phần T_1 có độ phức tạp là $T_1(n)=O(f(n))$
  - Phần T_2 có độ phức tạp là $T_2(n)=O(g(n))$
  - Thì độ phức tạp thuật toán là:

$$T(n)=T_1(n)+T_2(n) = O(f(n)+g(n))$$

- Chứng minh:

Các quy tắc của độ phức tạp

- Quy tắc Max: Nếu thuật toán T có độ phức tạp là $T(n)=O(f(n)+g(n))$ thì có thể coi thuật toán T có độ phức tạp là

$$T(n)=O(\max(f(n), g(n)))$$

- Chứng minh:

Các quy tắc của độ phức tạp

- Quy tắc Nhân: Nếu thuật toán T có độ phức tạp tính toán là $T(n)=O(f(n))$. Khi đó nếu thực hiện $k(n)$ lần thuật toán T với $k(n)=O(g(n))$ thì độ phức tạp tính toán là

$$O(f(n).g(n))$$

- Chứng minh:

Một số dạng hàm kí hiệu độ phức tạp thuật toán

- Tùy theo dạng hàm f(n), ta có các kí pháp sau:
  - Nếu thuật toán có thời gian thực hiện không phụ thuộc vào kích thước dữ liệu thì ta nói thuật toán có độ phức tạp là một hằng số và được viết là $O(1)$
  - Nếu thuật toán có thời gian thực hiện là $\log_a f(n)$ thì độ phức tạp của thuật toán đó được viết là $O(\log f(n))$
  - Nếu thuật toán có thời gian thực hiện là đa thức bậc k: P(n) thì độ phức tạp của thuật toán đó được viết là $O(n^k)$

Một số dạng hàm kí hiệu độ phức tạp thuật toán

Kí hiệu O-lớn	Tên thường gọi
O(1)	Hằng
O(log(n))	Logarit
O(n)	Tuyến tính
O(n.log(n))	n.log(n)
$O(n^2)$, $O(n^3)$, …	Bình phương, lập phương, … Đa thức
$O(2^n)$, $O(a^n)$	Mũ
O(n!)	Giai thừa

ƯỚC LƯỢNG ĐỘ PHỨC TẠP CỦA THUẬT TOÁN

Phân loại câu lệnh trong một ngôn ngữ lập trình

- Câu lệnh đơn thực hiện một thao tác
  - Lệnh gán đơn giản (không chứa lời gọi hàm trong biểu thức)
  - Đọc/ghi đơn giản
  - Câu lệnh chuyển điều khiển đơn giản (break, goto, continue, return)
- Câu lệnh hợp thành: dãy các câu lệnh trong 1 khối
- Câu lệnh rẽ nhánh: if, switch…case
- Câu lệnh lặp: for, while, do…while

Đánh giá độ phức tạp của từng câu lệnh

- Độ phức tạp của lệnh đơn:
  Không phụ thuộc kích thước dữ liệu nên sẽ là O(1)
- Độ phức tạp của lệnh hợp thành:
  Tính theo quy tắc Cộng và quy tắc Max
- Độ phức tạp của lệnh rẽ nhánh đầy đủ:
  Tính theo quy tắc Max. Nếu thời gian thực hiện 2 thành phần là f(n), g(n) thì độ phức tạp O(max(f(n), g(n)))
- Độ phức tạp của lệnh lặp:
  Tính theo quy tắc nhân

Một số ví dụ

- Ví dụ [for]: Tìm độ phức tạp của thuật toán

```
// Thuật toán tính tổng S=a[0]+a[1]+…+a[n-1]
{1} cout << "n: ";
{2} cin >> n;
{3} for (i=0; i<n; i++)
{4}   cin >> a[i];
{5} s = 0;
{6} for (i=0; i<n; i++)
{7}   s = s + a[i];
```

Một số ví dụ

- Ví dụ [for lồng nhau]: Tìm độ phức tạp của thuật toán

```
{1} s1 = 0;
{2} for (i=0; i<n; i++)
{3}   s1=s1+a[i];

{4} s2 = 0;
{5} for (i=0; i<n; i++)
{6}   for (j=0; j<n; j++)
{7}         s2 = s2 + b[i][j];
```

Một số ví dụ

- Ví dụ [for lồng nhau]: Tìm độ phức tạp của thuật toán

```
{1} s = 0;
{2} for (i=0; i<n; i++)
{3}   for (j=0; j<i; j++)
{4}         s = s + b[i][j];
```

Một số ví dụ

- Ví dụ [if]: Tìm độ phức tạp của thuật toán trong trường hợp xấu nhất

```
// Thuật toán tìm max
{1} max = a[0];
{2} for (i=1; i<n; i++)
{3}    if (max < a[i])
{4}          max=a[i];
```

Một số ví dụ

- Ví dụ [if + return]: Tìm độ phức tạp của thuật toán trong trường hợp xấu nhất

```
// Thuật toán tìm kiếm
{1} vitri = -1;
{2} for (i=0; i<n; i++)
{3}     if (x == a[i])
        {
{4}           vitri = i;
{5}           return vitri;
        }
{6} return vitri;
```

Một số ví dụ

- Ví dụ: Tìm độ phức tạp thuật toán Tính tổng ma trận

```
i=0;
s=0;
while (i<=n)
{
        j=1;
        while (j<=n)
        {
                s = s + a[i][j];
                j++;
        }
        i++;
}
```

Một số ví dụ

- Ví dụ:

```
smax=a[0];
i=0;
while (i<=n)
{
        j=i;
        while (j<=n)
        {
                k=i;
                s=0;
                while (k<=j)
                {
                        s = s + a[k];
                        k++;
                }
                if (s > smax)
                        s = smax;
                j++;
        }
        i++;
}
```

Tóm tắt chương 3

Chương 2

ÔN TẬP KỸ THUẬT XỬ LÝ FILE – MẢNG – XÂU KÝ TỰ

Nội dung

- Kỹ thuật xử lý file văn bản
- Kỹ thuật xử lý mảng
- Kỹ thuật xử lý xâu ký tự

Kỹ thuật xử lý file văn bản

- Thư viện

```
using System.IO;
using System.Diagnostics;
```

- Lớp

```
StreamReader
StreamWriter
```

Kỹ thuật xử lý file văn bản

- Ghi dữ liệu Text ra file
  - Tạo đối tượng stream-writer và mở file

```
StreamWriter sw = new StreamWriter("file");
```

  - Ghi dữ liệu ra file

```
sw.Write(value);
Sw.WriteLine(value);
```

  - Đóng file

```
sw.Close();
```

Kỹ thuật xử lý file văn bản

- Đọc dữ liệu Text từ file
  - Tạo đối tượng stream-reader và mở file

```
StreamReader sr = new StreamReader("file");
```

  - Đọc dữ liệu trong file

```
string s = sr.ReadLine();
string s = sr.ReadToEnd();
```

  - Đóng file

```
sr.Close();
```

Kỹ thuật xử lý file văn bản

- Ví dụ:

Kỹ thuật xử lý mảng

- Khai báo mảng

```
int[] a = new int[n];
int[,] a = new int[n,m];
```

- Sử dụng mảng

```
a[…] = …
a[…,…] = …
```

Kỹ thuật xử lý mảng

- Một số thuật toán cơ bản
  - Thuật toán Sắp xếp (Sort)
    - Sắp xếp chọn (Selection Sort)
    - Sắp xếp nhanh (Quicksort)
    - Sắp xếp phân bố (Distribution sort)
    - Sắp xếp theo chỉ mục
  - Thuật toán Tìm kiếm (Search)
    - Tìm kiếm tuyến tính
    - Tìm kiếm nhị phân

Kỹ thuật xử lý mảng

- Một số định hướng để thiết kế thuật toán hiệu qủa dựa trên kích thước bộ dữ liệu
  - Gọi N là kích thước của bộ dữ liệu
    - N≤200, dùng tối đa 4 for
    - N ≤ 1.000, dùng tối đa 3 for
    - N ≤ 40.000, dùng tối đa 2 for
    - Ngược lại, dùng tối đa 1 for

Kỹ thuật xử lý xâu ký tự

- Khai báo xâu

```
string s;
```

- Một số thuộc tính/phương thức trên xâu ký tự

```
int len = s.Length;

s = s.Insert(startIndex, value);
s = s.Remove(startIndex, count);
s = s.Replace(oldString, newString);

s = string.Format("format string", …);
```

Kỹ thuật xử lý xâu ký tự

- StringBuilder

```
StringBuilder sb;
```

- StringBuilder và string

```
string s;
…
StringBuilder sb = new StringBuilder(s);
…
s = sb.ToString();
```

Kỹ thuật xử lý xâu ký tự

- Một số thuộc tính/phương thức trên StringBuilder

```
sb.Insert(index, value);
sb.Remove(startIndex, length);
sb.Replace(oldString, newString);
sb.Append(value);
```

Kỹ thuật xử lý xâu ký tự

- Ví dụ 1: Lặp qua một đoạn ký tự liên tục
- Ví dụ 2: Kiểm tra ký tự là ký tự số
- Ví dụ 3: Kiểm tra chữ HOA

Tóm tắt chương 2

Chương 3

LẬP TRÌNH ĐỆ QUY

Nội dung

- Định nghĩa theo cách đệ quy
- Cài đặt Hàm đệ quy
- Hoạt động của Hàm đệ quy
- Phân loại đệ quy
- Ứng dụng của đệ quy
- Ưu điểm và khuyết điểm của đệ quy
- Một số phương pháp khử đệ quy
- Bài tập áp dụng

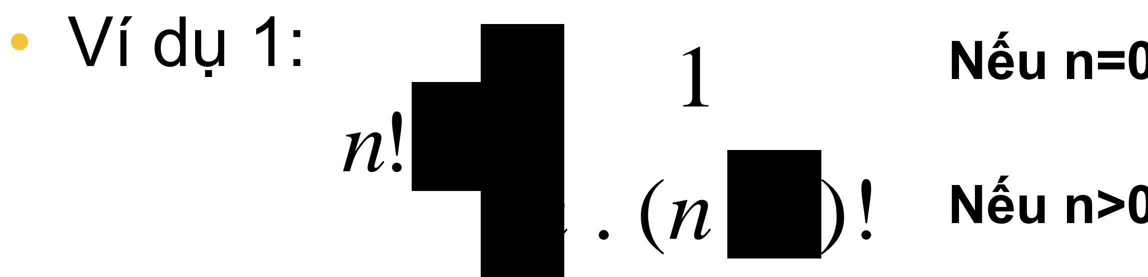
Định nghĩa theo cách đệ quy

- Định nghĩa theo cách đệ quy: Định nghĩa theo cách đệ quy của một khái niệm là định nghĩa khái niệm mới đó thông qua chính khái niệm đang muốn định nghĩa.

- Ví dụ: Định nghĩa tập số tự nhiên N
  - $0 \in N$
  - Nếu $n \in N$ thì $n+1 \in N$

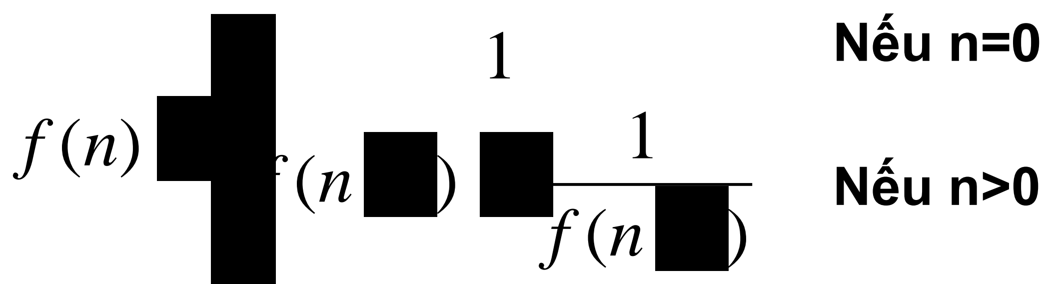
Định nghĩa theo cách đệ quy

- Mục đích của đệ quy:
  - Tạo ra các phần tử mới
  - Kiểm tra một phần tử có thuộc tập đã cho hay không
- Dùng định nghĩa theo cách đệ quy để định nghĩa các hàm hay chuỗi số (Hàm đệ quy, công thức đệ quy)
  - Ví dụ 1:



Định nghĩa theo cách đệ quy

- Ví dụ 2:



- Ví dụ 3: Công thức tính số Fibonacci

$$f(n) = \begin{cases} 1 & \text{Nếu } n=1 \text{ hay } n=2 \\ f(n-1) + f(n-2) & \text{Nếu } n>2 \end{cases}$$

Định nghĩa theo cách đệ quy

- Các thành phần của 1 định nghĩa theo cách đệ quy
  - Thành phần 1: Thành phần không đệ quy (trường hợp cơ bản, trường hợp cơ sở, trường hợp suy biến, điều kiện dừng)
    - Chứa những trường hợp đơn giản nhất để xây dựng nên tập hợp
  - Thành phần 2: Thành phần đệ quy (trường hợp đệ quy)
    - Chứa những quy tắc, công thức để tạo đối tượng mới từ những đối tượng trước đó

- Nhận xét: Thành phần đệ quy phải tiến về thành phần không đệ quy

Định nghĩa theo cách đệ quy

- Làm thế nào để tìm công thức đệ quy?
  - Chia bài toán f(n) thành các bài toán con f(1), f(2), …, f(n-1) có dạng giống bài toán f(n)
  - Tìm mối quan hệ giữa bài toán lớn với bài toán con
- Vấn đề khó khăn
  - Bao nhiêu bài toán con?
  - Chọn bài toán con nào?

Định nghĩa theo cách đệ quy

- Các bước gợi ý tìm công thức đệ quy f(n)
  - B1: Chọn một bài toán con f(k) (thường là f(n-1), f(n-2))
  - B2: Tìm mối quan hệ giữa f(n) với f(k)
  - B3: Nếu tìm được mối quan hệ thì
    Tìm trường hợp cơ sở
    Nhảy đến B5
  - B4: Ngược lại quay về B1 chọn bài toán con khác, nếu thấy không khả quan thì chọn một số bài toán con
  - B5: Kết thúc

Định nghĩa theo cách đệ quy

- Tìm định nghĩa đệ quy để tính tổng/tích của mảng số nguyên a có n phần tử ($n \leq 100$)

Định nghĩa theo cách đệ quy

- Tìm định nghĩa đệ quy để kiểm tra số nguyên n là số chẵn hay số lẻ (không dùng phép toán % và &)

Định nghĩa theo cách đệ quy

- Tìm định nghĩa đệ quy để tính độ dài của chuỗi s

Định nghĩa theo cách đệ quy

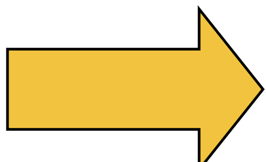
- Tìm định nghĩa đệ quy để kiểm tra chuỗi s có chứa ký tự ch không

Định nghĩa theo cách đệ quy

- Tìm định nghĩa đệ quy để đảo mảng số nguyên a có n phần tử ($n \leq 100$)

Định nghĩa theo cách đệ quy

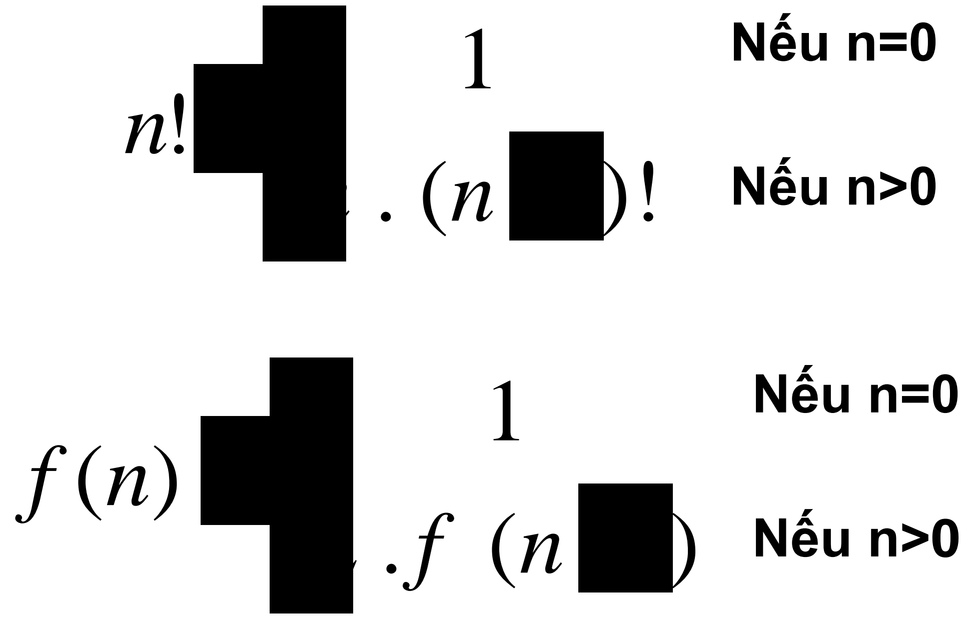
- Hạn chế của định nghĩa Đệ quy
  - Để tính f(n), chúng ta phải tính một vài hay tất cả các phần tử trước đó f(1), f(2), …
  - Để kiểm tra x có thuộc tập hợp đang định nghĩa hay không chúng ta cũng phải tính f(1), f(2), …



Tìm định nghĩa không đệ quy tương đương

Định nghĩa theo cách đệ quy

- Tìm định nghĩa không đệ quy của công thức đệ quy sau:



Cài đặt Hàm đệ quy

- Hàm/thủ tục Đệ quy: Một hàm A được gọi là Hàm Đệ quy nếu trong định nghĩa hàm A có lời gọi đến chính hàm A

```
KieuTraVe TenHam(Danh Sach Tham So)
{
        …
        TenHam();
        …
}
```

Cài đặt Hàm đệ quy

- Sơ đồ cài đặt
  - Sơ đồ 1:

```
KieuTraVe TenHam(Kieu n)
{
      if (dieukien dung)
            [return] kq;
      else
            [return] TenHam(n-1) …
}
```

Cài đặt Hàm đệ quy

- Sơ đồ cài đặt
  - Sơ đồ 2:

```
KieuTraVe TenHam(Kieu n)
{
      if (dieukien dequy)
            [return] TenHam(n-1)…;
      else
            [return] kq;
}
```

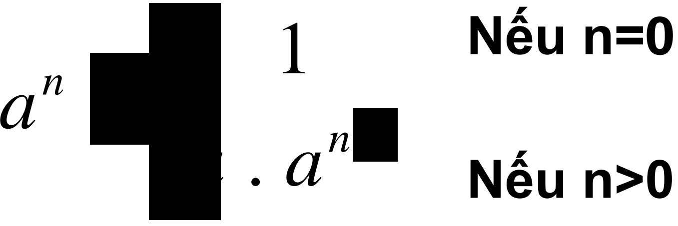
Cài đặt Hàm đệ quy

- Ví dụ: Cài đặt hàm đệ quy tính n!

```
int Factorial(int n)
{








}
```

Hoạt động của Hàm đệ quy

- Tìm hiểu hoạt động của hàm đệ quy tính a^n



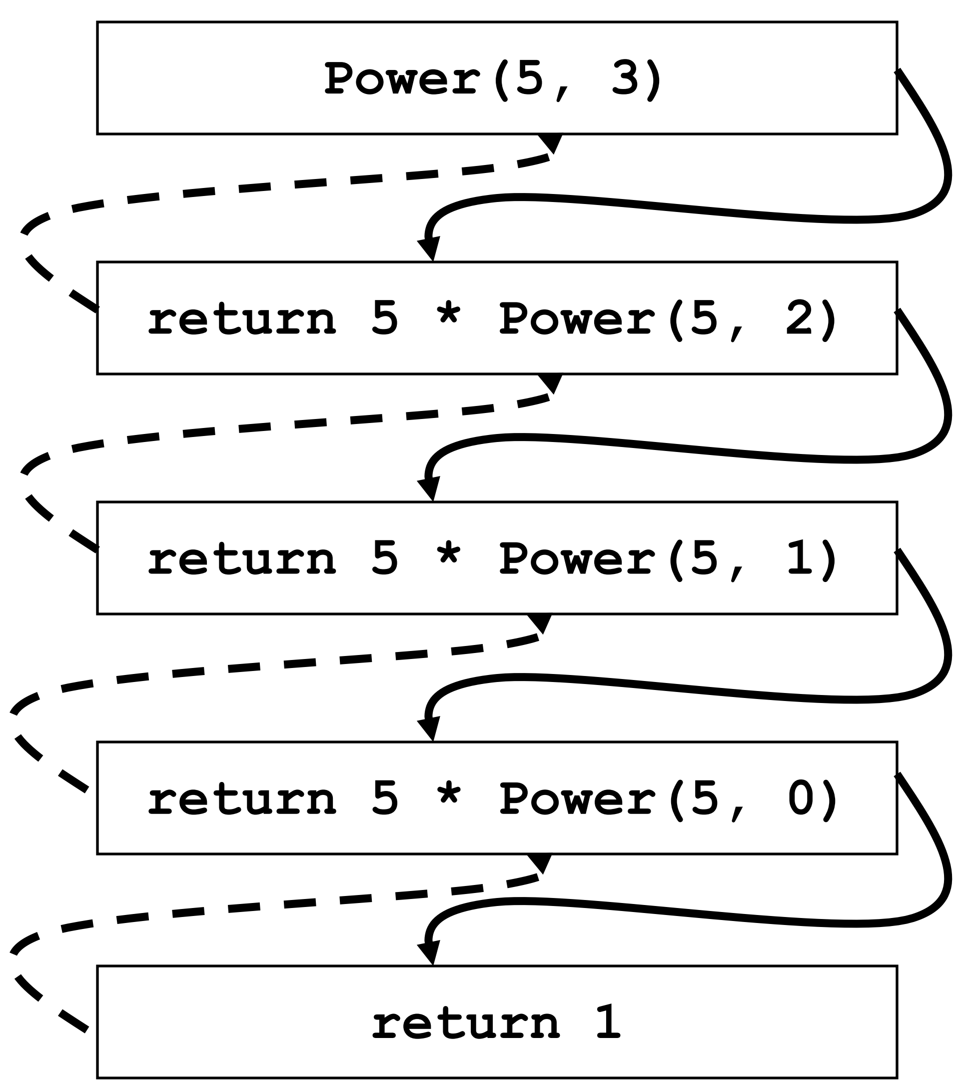

```
double Power(double a, int n)
{



}
```

Hoạt động của Hàm đệ quy

- Phân tích hoạt động của hàm Power(a, n) một cách hình thức:
  - Gồm 2 pha:
    - Pha tiến (forward): Tiến đến lời giải nhỏ nhất
    - Pha lùi (backward): Kết hợp các kết quả lại với nhau

  - Ví dụ: Cho a= 5, n=3, hãy theo vết chạy của hàm Power(5, 3)

Hoạt động của Hàm đệ quy



Hoạt động của Hàm đệ quy

- Hoạt động của hệ thống:
  - Hệ thống lưu giữ trạng thái của tất cả các lời gọi hàm trong ngăn xếp
  - Mỗi khi có một lời gọi hàm, hệ thống tạo ra 1 vùng lưu trữ trong ngăn xếp gọi là bản ghi kích hoạt (activation record) để lưu thông tin
    - Giá trị trả về
    - Địa chỉ trả về
    - Địa chỉ các liên kết động
    - Tham số truyền vào
    - Các biến cục bộ

Hoạt động của Hàm đệ quy

Phân loại đệ quy

- Đệ quy có thể được phân thành một số trường hợp sau:
  - Đệ quy trực tiếp
  - Đệ quy gián tiếp
  - Đệ quy tuyến tính
  - Đệ quy nhánh (đệ quy không tuyến tính, đệ quy cây)
  - Đệ quy đuôi
  - Đệ quy lồng nhau

Phân loại đệ quy
Đệ quy trực tiếp

- Định nghĩa [Đệ quy trực tiếp – Direct Recursion]: Một hàm được gọi là Hàm Đệ quy trực tiếp nếu hàm đó có lời gọi đến chính nó một cách rõ ràng

- Ví dụ:

```
int Foo (int x)
{
    if (x <= 0)  return x;

    return Foo(x - 1);
}
```

Phân loại đệ quy
Đệ quy gián tiếp

- Định nghĩa [Đệ quy gián tiếp – Indirect Recursion]: Một hàm được gọi là Hàm Đệ quy gián tiếp nếu hàm đó gọi đến nó thông qua những lời gọi hàm khác

- Sơ đồ

  $f() \rightarrow g_1() \rightarrow g_2() \rightarrow \ldots \rightarrow g_n() \rightarrow f()$

Phân loại đệ quy
Đệ quy gián tiếp

- Ví dụ:

```
int Foo(int x)
{
    if (x <= 0)
        return x;

   return Bar(x);
}
```

```
int Bar(int y)
{
     return Foo(y - 1);
}
```

Phân loại đệ quy
Đệ quy tuyến tính

- Định nghĩa [Đệ quy tuyến tính – Linear Recursion]: Một hàm đệ quy được gọi là đệ quy tuyến tính nếu hàm đó không có toán tử phụ thuộc vào 2 lời gọi đệ quy trở lên

```
int Factorial (int n)
{
    if (n == 0)
      return 1;

    return n * Factorial(n - 1);
}
```

Phân loại đệ quy
Đệ quy nhánh

- Định nghĩa [Đệ quy nhánh – Tree Recursion]: Một hàm đệ quy được gọi là đệ quy nhánh nếu hàm đó có toán tử phụ thuộc vào 2 lời gọi đệ quy trở lên

```
int Fibonacci (int n)
{
    if (n==1 || n==2)
      return 1;

    return Fibonacci(n - 1) + Fibonacci(n-2);
}
```

Phân loại đệ quy
Đệ quy đuôi

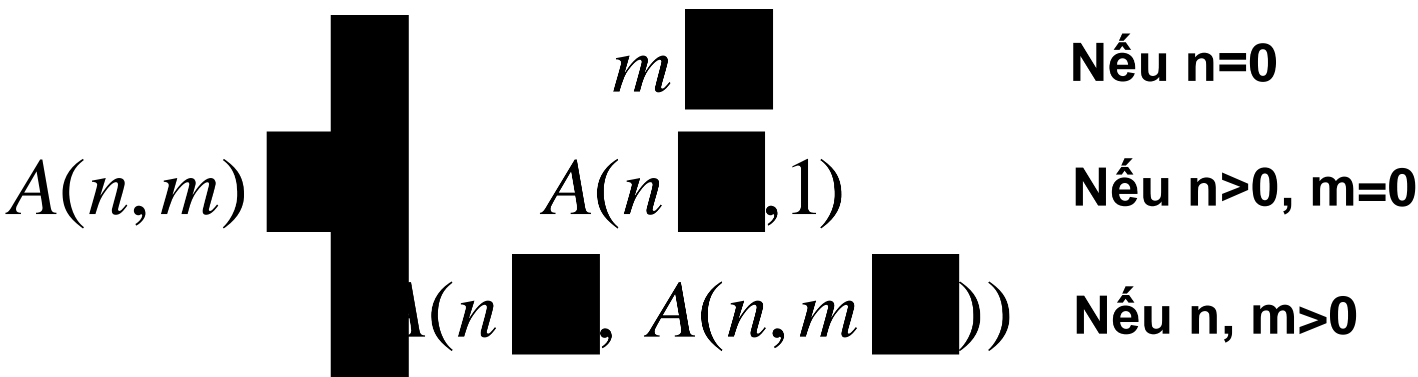
- Định nghĩa [Đệ quy đuôi – Tail Recursion]: Hàm Đệ quy đuôi là một hàm đệ quy thỏa:
  - Chứa duy nhất 1 lời gọi đệ quy
  - Lời gọi đệ quy nằm ở cuối hàm
  - Lời gọi đệ quy trước không phụ thuộc lời gọi đệ quy sau
- Ví dụ:

```
void InSo(int n)
{
        if (n>0)
        {
                cout << n;
                InSo(n-1);
        }
}
```

Phân loại đệ quy
Đệ quy lồng nhau

- Định nghĩa [Đệ quy lồng nhau]: Hàm Đệ quy lồng nhau là hàm gọi đến chính nó và sử dụng chính hàm đó như là tham số đầu vào của hàm

- Ví dụ:



Ứng dụng của đệ quy

- Lập trình đệ quy được dùng trong một số trường hợp sau
  - Dùng trong phương pháp chia để trị
  - Dùng trong phương pháp quy hoạch động
  - Dùng trong phương pháp quay lui vét cạn
  - …

Ưu điểm và khuyết điểm của đệ quy

- Ưu điểm
  - Trong sáng
  - Dễ đọc
  - Cài đặt đơn giản, ngắn gọn
- Khuyết điểm
  - Phải lưu nhiều trạng thái trong stack: Có thể tràn ngăn xếp
  - Làm chậm thời gian thực thi chương trình

Một số phương pháp khử đệ quy

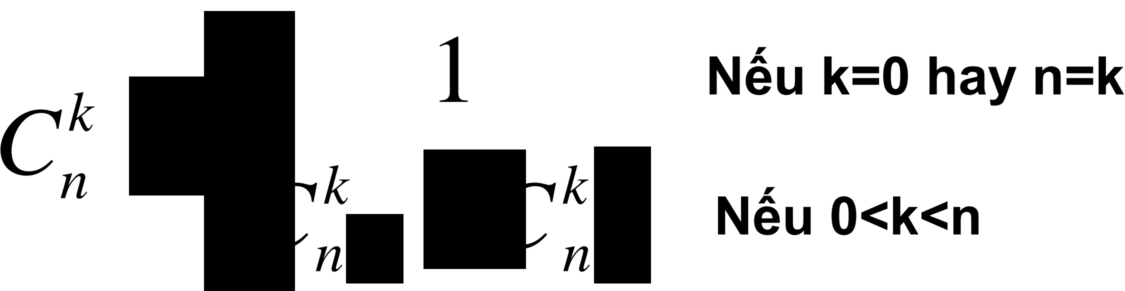
- Một số gợi ý:
  - Tìm công thức không đệ quy
  - Dùng mảng lưu trữ dữ liệu trung gian
  - Dùng stack để mô phỏng đệ quy
  - …

Bài tập áp dụng

- Viết hàm đệ quy Tính Ước số chung lớn nhất của a và b (USCLN(a, b))

$$USCLN(a,b) = \begin{cases} a & \textbf{Nếu b=0} \\ USCLN(b, a \bmod b) & \textbf{Nếu b≠0} \end{cases}$$

- Viết hàm đệ quy tính C_n^k



Bài tập áp dụng

- Viết hàm đệ quy In mảng a gồm n phần tử ($n \le 100$) lên màn hình
- Viết hàm đệ quy In ra các chữ số của số nguyên n theo thứ tự đảo ngược
- Viết hàm đệ quy Tìm số lớn nhất /nhỏ nhất của mảng số nguyên a có n phần tử ($n \le 100$)
- Viết hàm đệ quy Đếm số lần xuất hiện của ký tự ch trong chuỗi s

Bài tập áp dụng

- Viết hàm đệ quy Kiểm tra n có phải là số nguyên tố không
- Viết hàm đệ quy Giải bài toán tháp Hanoi
- Viết hàm đệ quy liệt kê các phân số tối giản không giảm nằm trong [0, 1] và có mẫu số nhỏ hơn hay bằng n
- Viết hàm đệ quy Tính giá trị của một số viết dưới dạng chữ LA MÃ

Tóm tắt chương 4

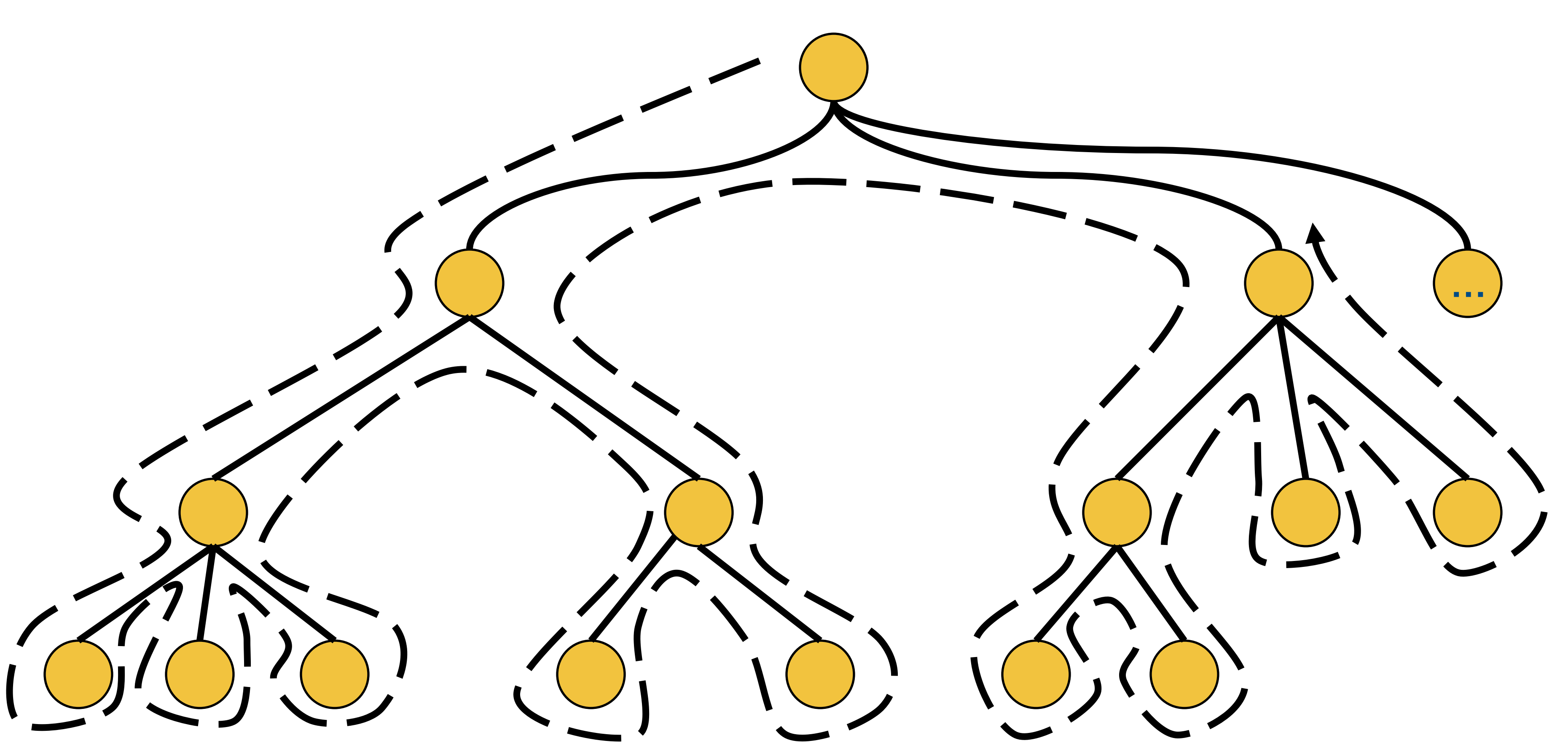
Chương 4

PHƯƠNG PHÁP THIẾT KẾ THUẬT TOÁN – QUAY LUI –

Nội dung

- Giới thiệu
- Phương pháp
- Sơ đồ cài đặt
- Các ví dụ
- Ưu điểm và khuyết điểm

Hình ảnh



Giới thiệu

- Định nghĩa [Quay lui – Backtracking]:
  - Quay lui là một phương pháp thiết kế thuật toán để tìm nghiệm của bài toán bằng cách xét tất cả các phương án.
  - Một phương án gồm nhiều thành phần, và phương pháp quay lui sẽ xây dựng từng thành phần trong mỗi bước.
  - Trong quá trình xây dựng thành phần thứ i (tìm nghiệm cho thành phần thứ i), nếu không thể xây dựng được thì quay lại chọn nghiệm khác cho thành phần thứ (i-1)

Bài toán

- Phát biểu bài toán: Giả sử nghiệm của bài toán cần tìm có dạng $X=(x_1, x_2, \ldots, x_k, \ldots)$, trong đó
  - x_i là 1 thành phần nghiệm của bài toán
  - x_i có một miền giá trị D_i nào đó $(x_i \in D_i)$.
  - Số lượng thành phần x_i có thể xác định hay không xác định
  - Bài toán có những ràng buộc là F
- Yêu cầu: Hãy xây dựng 1 nghiệm hay tất cả các nghiệm của bài toán thỏa điều kiện F

Phương pháp

- Phương pháp Quay lui
  - Phương pháp Quay lui xây dựng dần nghiệm X của bài toán: Bắt đầu từ x_1 được chọn ra từ tập D_1, rồi đến x_2 được chọn ra từ tập D_2, ... bằng cách thử mọi khả năng có thể xảy ra.

  - Một cách tổng quát: Nếu chúng ta đã xác định được lời giải bộ phận gồm (i-1) thành phần $X(i-1) = (x_1, x_2, ..., x_{i-1})$, bây giờ chúng ta tìm giá trị cho thành phần x_i bằng cách xét mọi khả năng có thể có của x_i trong tập D_i. Với mỗi khả năng j (j[illegible]$_i$), chúng ta kiểm tra xem có thể thỏa điều kiện là nghiệm thành phần của bài toán không

Phương pháp

- Có 2 khả năng xảy ra:
  - Nếu khả năng j thỏa điều kiện thì
    - Gán $x_i = j$
    - Tiếp theo tìm nghiệm cho thành phần x_{i+1}
  - Nếu đã thử mọi khả năng của j mà không thỏa điều kiện bài toán thì có nghĩa là đi theo con đường

    $$X(i-1) = (x_1, x_2, \ldots, x_{i-1})$$

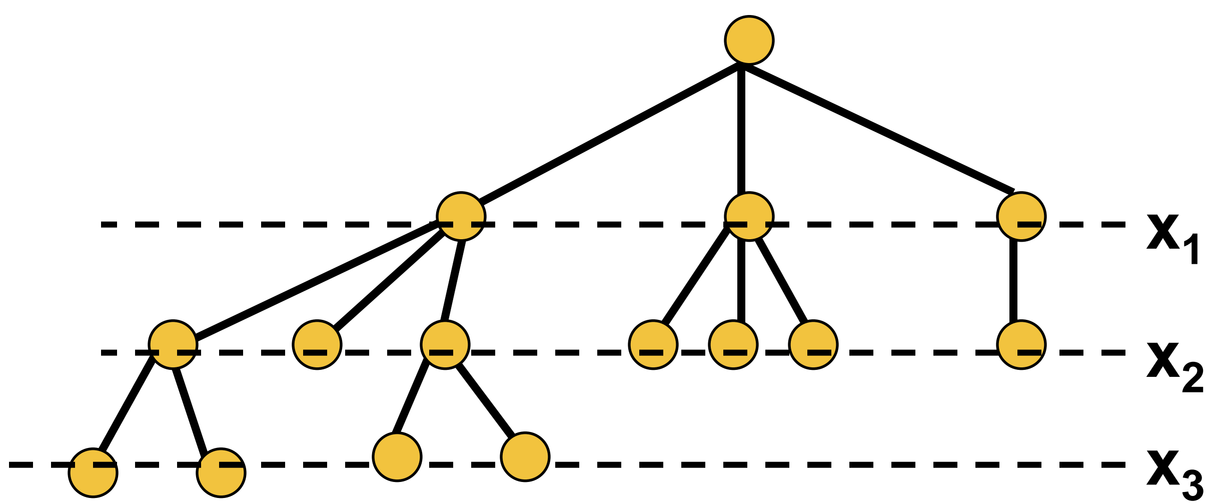
    sẽ không thể dẫn đến kết quả. Chúng ta quay về bước trước để xác định lại x_{i-1} (bằng cách chọn 1 giá trị khác trong D_{i-1}).

Phương pháp

- Quá trình này dừng cho đến khi tìm được nghiệm của bài toán hay vét qua hết khả năng mà không thể tìm được nghiệm của bài toán

Phương pháp

- Cây tìm kiếm (Cây không gian tìm kiếm): Quá trình tìm kiếm lời giải theo phương pháp Quay lui sẽ sinh ra 1 cây tìm kiếm



Phương pháp

- Đặc điểm của phương pháp Quay lui
  - Xây dựng dần từng thành phần trong 1 phương án
  - Trong quá trình xây dựng phương án nó thực hiện:
    - Tiến: Để mở rộng các thành phần của phương án
    - Lui: Nếu không thể mở rộng phương án
  - Xét mọi khả năng có thể xảy ra
- Phương pháp quay lui còn được gọi với những tên khác như: Vét cạn (Exhaustion), Duyệt, thử và sai (Trial and Error), …

Phương pháp

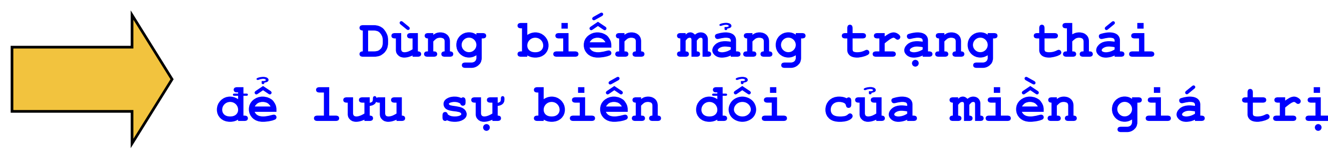
- Các bước sử dụng phương pháp Quay lui
  - Bước 1 [Biểu diễn nghiệm]: Biểu diễn nghiệm bài toán dưới dạng một vector

    $$X=(x_1, x_2, x_3, \ldots, x_k, \ldots)$$

  - Bước 2 [Tìm miền giá trị thô]: Xác định các miền giá trị cơ bản D_i cho các x_i (D_i=[min_i, max_i])
  - Bước 3 [Ràng buộc]: Tìm những ràng buộc của x_i và giữa x_i và x_j. Từ đó có thể xác định lại các D_i
  - Bước 4: Xác định những điều kiện F khác để X là nghiệm của bài toán

Phương pháp

- Xác định miền giá trị D_i (Bước 3):
  - Xác định cận trên và cận dưới của miền D_i (D_i=[min_i, max_i])
  - Chi tiết việc xác định D_i
    - Nếu các D_i và D_j độc lập nhau thì không cần chỉnh sửa D_i trong bước 2
    - Nếu D_i bị thay đổi do việc chọn lựa ở những thành phần x_j (j<i) trước đó thì chúng ta cần xác định lại D_i khi chọn lựa x_j ở bước j



Sơ đồ cài đặt

- Nếu các D_i và D_j độc lập nhau:

```
void BackTrack_1A(int i)
{
	if (thỏa điều kiện bài toán F)
		Tìm được 1 nghiệm
	else
		for (j [illegible] D_i)
		{
			x_i = j;
			BackTrack_1A(i+1);
		}
}
```

Sơ đồ cài đặt

- Nếu các D_i và D_j độc lập nhau:

```
void BackTrack_1B(int i)
{
        for (j ∈ Di)
        {
                xi = j;
                if (thỏa điều kiện bài toán F)
                        Tìm được 1 nghiệm
                else
                        BackTrack_1B(i+1);
        }
}
```

Sơ đồ cài đặt

- Nếu các D_i và D_j phụ thuộc nhau:

```
void BackTrack_2A(int i)
{
        if (thỏa điều kiện bài toán F)
              Tìm được 1 nghiệm
        else
              for (j ■Di và status[j]==0)
              {
                    status[j] = 1;
                    xi = j;
                    BackTrack_2A(i+1);
                    status[j]=0;
              }
}
```

Sơ đồ cài đặt

- Nếu các D_i và D_j phụ thuộc nhau :

```
void BackTrack_2B(int i)
{
        for (j [illegible]Di và status[j]==0)
        {
                status[j]=1;
                xi = j;
                if (thỏa điều kiện bài toán F)
                        Tìm được 1 nghiệm
                else
                        BackTrack_2B(i+1);
                status[j]=0;
        }
}
```

Sơ đồ cài đặt

- Sơ đồ tổng quát

```
void BackTrack_3A(int x[], int i, data input)
{
	int D[MAXCANDIDATES];
	int nD;

	if (IsSolution(x, i))
		ProcessSolution(x, i, input);
	else
	{
		ConstructCandidates(x, i, input, D, &nD);
		for (j [illegible]D)
		{
			x[i] = j;
			BackTrack_3A(x, i+1, input);
		}
	}
}
```

Sơ đồ cài đặt

- Sơ đồ tổng quát

```
void BackTrack_3B(int x[], int i, data input)
{
        int D[MAXCANDIDATES];
        int nD;

        ConstructCandidates(x, i, input, D, &nD);
        for (j [illegible]D)
        {
                x[i] = j;
                if (IsSolution(x, i, input))
                        ProcessSolution(x, I, input);
                else
                        BackTrack_3B(x, i+1, input);
        }
}
```

Các ví dụ: {1} Tổ hợp

- Một tổ hợp chập k của tập n phần tử (k≤n) là một tập hợp con gồm k phần tử của tập n phần tử
  - Ví dụ: Tập {1, 2, 3, 4, 5} có các tổ hợp chập 2 là:
  - Số lượng tổ hợp chập k của tập n phần tử:

Các ví dụ: {1} Tổ hợp

- Bài toán: Hãy tìm tất cả các tổ hợp chập k của tập n phần tử
  - Bước 1: Biểu diễn nghiệm X
  - Bước 2: Tìm miền giá trị D_i của x_i
  - Bước 3: Ràng buộc giữa x_i và x_j
  - Bước 4: Xác định điều kiện F để X là nghiệm

Các ví dụ: {1} Tổ hợp

- Cấu trúc dữ liệu

```
#define MAX 20

int x[MAX+1];
int n, k;
```

Các ví dụ: {1} Tổ hợp

```
void ToHop(int i)
{



}
```

Các ví dụ: {2} Chỉnh hợp lặp

- Một chỉnh hợp lặp chập k của tập n phần tử (k≤n) là một bộ (có thứ tự) gồm k phần tử của tập n phần tử và các thành phần của bộ có thể trùng nhau
  - Ví dụ: Tập {1, 2, 3, 4, 5} có các chỉnh hợp lặp chập 2 là:
  - Số lượng chỉnh hợp lặp chập k của tập n phần tử:

$$\overline{A}_n^k \blacksquare n^k$$

Các ví dụ: {2} Chỉnh hợp lặp

- Bài toán: Hãy tìm tất cả các chỉnh hợp lặp chập k của tập n phần tử
  - Bước 1: Biểu diễn nghiệm X
  - Bước 2: Tìm miền giá trị D_i của x_i
  - Bước 3: Ràng buộc giữa x_i và x_j
  - Bước 4: Xác định điều kiện F để X là nghiệm

Các ví dụ: {2} Chỉnh hợp lặp

- Cấu trúc dữ liệu

```
#define MAX 20

int x[MAX+1];
int n, k;
```

Các ví dụ: {2} Chỉnh hợp lặp

```
void ChinhHopLap(int i)
{



}
```

Các ví dụ: {3} Chỉnh hợp không lặp

- Một chỉnh hợp không lặp chập k của tập n phần tử (k≤n) là một bộ (có thứ tự) gồm k phần tử của tập n phần tử và các thành phần của bộ không được trùng nhau
  - Ví dụ: Tập {1, 2, 3, 4, 5} có các chỉnh hợp không lặp chập 2 là:
  - Số lượng chỉnh hợp không lặp chập k của tập n phần tử:

$$A_n^k = n(n-1)\ldots(n-k+1) = \frac{n!}{(n-k)!}$$

Các ví dụ: {3} Chỉnh hợp không lặp

- Bài toán: Hãy tìm tất cả các chỉnh hợp không lặp chập k của tập n phần tử
  - Bước 1: Biểu diễn nghiệm X
  - Bước 2: Tìm miền giá trị D_i của x_i
  - Bước 3: Ràng buộc giữa x_i và x_j
  - Bước 4: Xác định điều kiện F để X là nghiệm

Các ví dụ: {3} Chỉnh hợp không lặp

- Cấu trúc dữ liệu

```
#define MAX 20

int x[MAX+1];
int n, k;

int status[MAX+1];
```

Các ví dụ: {3} Chỉnh hợp không lặp

```
void ChinhHopKhongLap(int i)
{
}
```

Các ví dụ: {4} Xếp 8 Hậu

- Bài toán: Hãy đặt 8 con hậu lên bàn cờ vua 8x8, sao cho không con hậu nào được ăn con hậu nào, tức là chúng
  - Không cùng hàng
  - Không cùng cột
  - Không cùng đường chéo

Các ví dụ: {4} Xếp 8 Hậu

- Bước 1: Biểu diễn nghiệm X
- Bước 2: Tìm miền giá trị D_i của x_i
- Bước 3: Ràng buộc giữa x_i và x_j
- Bước 4: Xác định điều kiện F để X là nghiệm

Các ví dụ: {4} Xếp 8 Hậu

- Cấu trúc dữ liệu

```
#define MAX 8

int x[MAX+1];


int cot[MAX+1];
int cheoChinh[    ...    ];
int cheoPhu[    ...    ];
```

Các ví dụ: {4} Xếp 8 Hậu

```
void Xep8Hau(int i)
{









}
```

Ưu điểm và khuyết điểm

- Ưu điểm
  - Luôn đảm bảo tìm nghiêm đúng
  - Cho phép liệt kê mọi nghiệm của bài toán
- Khuyết điểm
  - Độ phức tạp thuật toán lớn
  - Thời gian thực thi lâu
  - Chỉ giải những bài toán có kích thước nhỏ

Tóm tắt chương 4

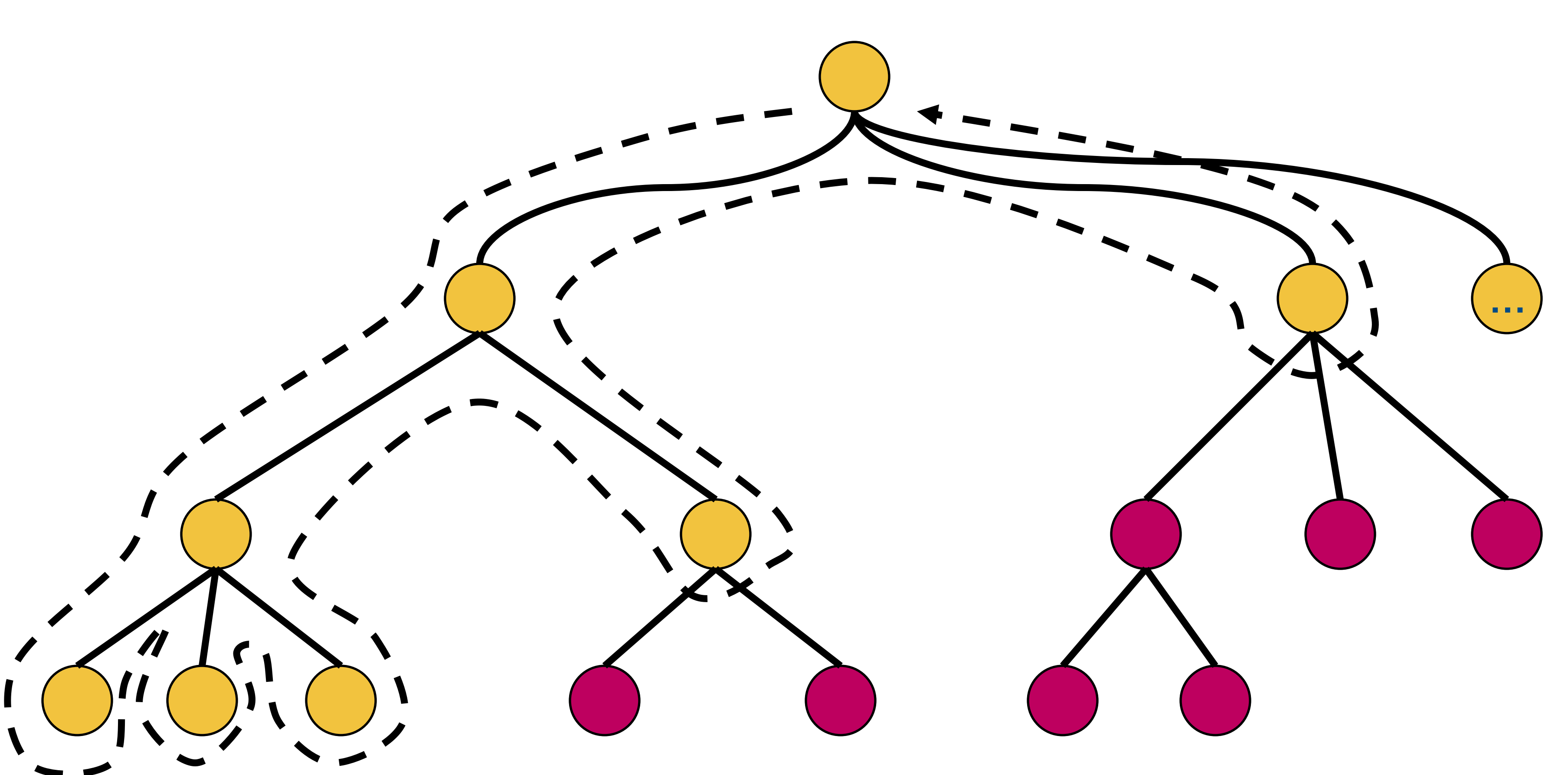
Chương 5

PHƯƠNG PHÁP THIẾT KẾ THUẬT TOÁN – NHÁNH CẬN –

Nội dung

- Giới thiệu
- Bài toán tối ưu
- Phương pháp
- Sơ đồ cài đặt
- Các ví dụ

Hình ảnh



Giới thiệu

- Bài toán tối ưu: Trong nhiều bài toán thực tế yêu cầu chúng tìm nghiệm thỏa mãn những điều kiện nào đó và nghiệm này phải tốt nhất theo tiêu chí cụ thể nào đó.

- Phương pháp Nhánh cận là một dạng cải tiến của phương pháp quay lui dùng để giải quyết bài toán tối ưu

Bài toán tối ưu

- Phát biểu bài toán: Giả sử bài toán yêu cầu tìm phương án $X=(x_1, x_2, \ldots, x_k, \ldots)$ thỏa mãn những điền kiện nào đó và phương án này là tốt nhất theo tiêu chí cụ thể nào đó.

  - Gọi f(X) là hàm đánh giá sự tốt nhất của phương án X (f là hàm mục tiêu hay hàm chi phí)

  - Yêu cầu: Tìm X sao cho

f(X) → min (max)

Bài toán tối ưu

- Nếu gọi X* là phương án tốt nhất (tối ưu)

$$X^* = \arg\min_X f(X)$$

$$X^* = \arg\max_X f(X)$$

- f* = f(X*) được gọi là giá trị tối ưu của bài toán

Bài toán tối ưu

- Ví dụ 1 [Bài toán người du lịch – Traveling Salesman Problem – TSP]
  Cho n thành phố được đánh số từ 1 đến n và khoảng cách giữa thành phố i và thành phố j được cho bởi c_{ij} (chú ý: $c_{ij}=c_{ji}$)

  Yêu cầu: Tìm một hành trình ngắn nhất cho phép viếng thăm n thành phố, mỗi thành phố viếng thăm đúng 1 lần và quay về thành phố ban đầu.

Bài toán tối ưu

- Mô hình toán học:
  - Một hành trình là 1 hoán vị X=(x(1), x(2), …, x(n)) của n số {1, 2, …, n}
  - Hàm mục tiêu:

$$f(X) = c_{x(1),x(2)} + c_{x(2),x(3)} + \dots + c_{x(n-1),x(n)} + c_{x(n),x(1)}$$

  - Yêu cầu:

$$X^{*} = \arg\min_{X} f(X)$$

Bài toán tối ưu

- Ví dụ 2 [Bài toán phân công – Job Assignment Problem – JAP]
  Có n công việc và n nhân viên. Gọi c_{ij} là chi phí để trả cho nhân viên i khi làm công việc j.

  Yêu cầu: Tìm cách phân công n nhân viên làm n việc trên sao cho tổng chi phí là nhỏ nhất (một nhân viên chỉ làm 1 việc, một việc chỉ do 1 nhân viên làm).

Bài toán tối ưu

- Mô hình toán học:
  - Một cách phân công n nhân viên làm n việc là 1 hoán vị X=(x(1), x(2), …, x(n)) của n số {1, 2, …, n}
  - Hàm mục tiêu:

$$f(X) = c_{1,x(1)} + c_{2,x(2)} + \dots + c_{n,x(n)}$$

  - Yêu cầu:

$$X^* = \arg\min_X f(X)$$

Bài toán tối ưu

- Ví dụ 3 [Bài toán cái túi – 0-1 Knapsack problem]
  Cho n loại đồ vật được đánh số từ 1 đến n, đồ vật thứ i có giá trị v_i và trọng lượng w_i.
  Cho trước cái túi có trọng lượng tối đa mà nó có thể mang là W.

  Yêu cầu: Tìm một số đồ vật để bỏ vào túi sao cho tổng trọng lượng các đồ vật bỏ vào túi không vượt quá W và tổng giá trị của các đồ vật là lớn nhất.

Bài toán tối ưu

- Nhận xét:
  - Hình thức thông thường nhất của bài toán là bài toán 0-1 knapsack, trong đó giới hạn số lượng x_i của loại đồ vật i là 0 hay 1

Bài toán tối ưu

- Mô hình toán học:
  - Một phương án chọn đồ vật được biểu diễn bằng 1 vector nhị phân độ dài n: $X=(x_1, x_2, \ldots, x_n)$. ($x_i$ [illegible], 1})
    - $x_i=1$: Chọn đồ vật i
    - $x_i=0$: Không chọn đồ vật i
  - Tổng giá trị của các đồ vật đã chọn

$$f(X) \text{ [illegible] } {}_1 x_1 \text{ [illegible] } {}_2 x_2 \text{ [illegible] } . \text{ [illegible] } {}_n x_n \text{ [illegible] } x_i$$

  - Tổng trọng lượng của các đồ vật đã chọn

$$g(X) \text{ [illegible] } w_1 x_1 \text{ [illegible] } w_2 x_2 \text{ [illegible] } . \text{ [illegible] } w_n x_n \text{ [illegible] } x_i$$

Bài toán tối ưu

- f là hàm mục tiêu phải thỏa mãn điều kiện hàm g

- Yêu cầu:

$$X^* = \arg\max_{X \text{ và } g(X) \in V} f(X)$$

Phương pháp

- Phương pháp Nhánh cận (Branch and bound – B&B)

$$X^* = \arg\min_{X} f(X)$$

  - Giả sử ta tìm được hàm $g(x_1, x_2, \ldots, x_k)$ là hàm cận dưới của nghiệm có k thành phần

$$g(x_1, x_2, \ldots, x_k) \leq \min\{f(X)\}$$

Phương pháp

- Bước 1 [Khởi tạo]: Dùng 2 biến X_t và F_t để lưu lại nghiệm tốt nhất trong quá trình tìm nghiệm ($F_t = f(X_t)$)
  - $X_t = ()$
  - $F_t = +$■
- Bước 2 [Quay lui]: Dùng phương pháp quy lui để xét tất cả các nghiệm có thể có của bài toán
  - Khi tìm được 1 nghiệm, ta so sánh f(X) với F_t. Nếu $F_t > f(X)$ thì ta lưu nghiệm tốt hơn lại.
    - $X_t = X$
    - $F_t = f(X)$

Phương pháp

- Bước 3 [Nhánh cận]:
  - Trong quá trình xây dựng nghiệm, giả sử đã xây dựng được nghiệm gồm k thành phần $X=(x_1, x_2, \ldots, x_k)$.
  - Bây giờ ta dự định mở rộng nghiệm thành $(x_1, x_2, \ldots, x_k, x_{k+1})$ nhưng nếu ta biết rằng những nghiệm mở rộng $(x_1, x_2, \ldots, x_k, x_{k+1}, \ldots)$ không thể tốt hơn F_t (nghĩa là $g(x_1, x_2, \ldots, x_k, x_{k+1}, \ldots) > F_t$) thì ta không cần mở rộng $(x_1, x_2, \ldots, x_k)$, chúng ta cắt đi những nghiệm (nhánh) không cần thiết.

Phương pháp

- Nhận xét
  - Phương pháp nhánh cận không quét qua toàn bộ các nghiệm có thể có của bài toán
  - Khó khăn của phương pháp nhánh cận là làm thế nào đánh giá được các nghiệm mở rộng (cận). Nếu đánh giá tốt sẽ bỏ nhiều nghiệm không cần thiết phải xét (nhánh)

Sơ đồ cài đặt

```
void BranchAndBound1(int i)
{
        if (thỏa điều kiện bài toán F)
        {
                Tìm được một nghiệm
                Cập nhật Xt và Ft
        }
        else
                for (j [illegible] Di)
                {
                        xi = j;
                        if (g(x1, x2, …, xi) < Ft)
                                BranchAndBound1(i+1);
                }
}
```

Sơ đồ cài đặt

```
void BranchAndBound2(int i)
{
        for (j [illegible] Di)
        {
                xi = j;
                if (thỏa điều kiện bài toán F)
                {
                        Tìm được một nghiệm
                        Cập nhật Xt và Ft
                }
                else
                        if (g(x1, x2, …, xi) < Ft)
                                BranchAndBound2(i+1);
        }
}
```

Ví dụ:

- Ví dụ [Bài toán người du lịch – Traveling Salesman Problem – TSP]
  - Mô hình toán:
    - Một hành trình là 1 hoán vị X=(x(1), x(2), …, x(n)) của n số {1, 2, …, n}
    - Hàm mục tiêu

$$f(X) = c_{x(1),x(2)} + c_{x(2),x(3)} + \cdots + c_{x(n-1),x(n)} + c_{x(n),x(1)}$$

  - Không mất tính tổng quát: Chúng ta có thể xuất phát từ thành phố số 1

$$f(X) = c_{1,x(2)} + c_{x(2),x(3)} + \cdots + c_{x(n-1),x(n)} + c_{x(n),1}$$

Ví dụ:

- Tìm cận dưới $g(x_1, x_2, \ldots, x_k)$
  - Cách 1: $g(x_1, x_2, \ldots, x_k) = f(x_1, x_2, \ldots, x_k)$

  - Thuật toán:
    - Bước 1 [Khởi động]: F_t=+■
    - Bước 2 [Quay lui]
    - Bước 3 [Nhánh cận]: Với mỗi bước thử x_i chúng ta kiểm tra độ dài đường đi đến lúc đó có nhỏ hơn F_t không. Nếu không nhỏ hơn thì chọn giá trị khác cho x_i

Ví dụ:

```
void TSP1(int i)
{

}
```

Ví dụ:

- Tìm cận dưới $g(x_1, x_2, \ldots, x_k)$
  - Cách 2:
    - Gọi $c_{min} = \min \{c_{ij}\}$ là độ dài của đoạn đường đi nhỏ nhất giữa các thành phố
    - Giả sử đã đi qua k thành phố $X=(x_1, x_2, \ldots, x_k)$. Độ dài đường đi qua k thành phố này là

$$T = f(X) = c_{1,x(2)} + c_{x(2),x(3)} + \ldots + c_{x(k-1),x(k)}$$

    - Cần phải đi qua (n-k) thành phố nữa, hay phải qua (n-k+1) đoạn đường, mỗi đoạn đường có khoảng cách không ít hơn c_{min}.
    - Cận dưới: $g(x_1, x_2, \ldots, x_k) = T + (n-k+1)c_{min}$

Ví dụ:

- Thuật toán:
  - Bước 1 [Khởi động]: F_t=+■
  - Bước 2 [Quay lui]
  - Bước 3 [Nhánh cận]: Với mỗi bước thử x_i chúng ta kiểm tra $T+(n-k+1)*c_{min}$ có nhỏ hơn F_t không. Nếu không nhỏ hơn thì chọn giá trị khác cho x_i

Ví dụ:

```
void TSP2(int i)
{
}
```

Tóm tắt chương 6

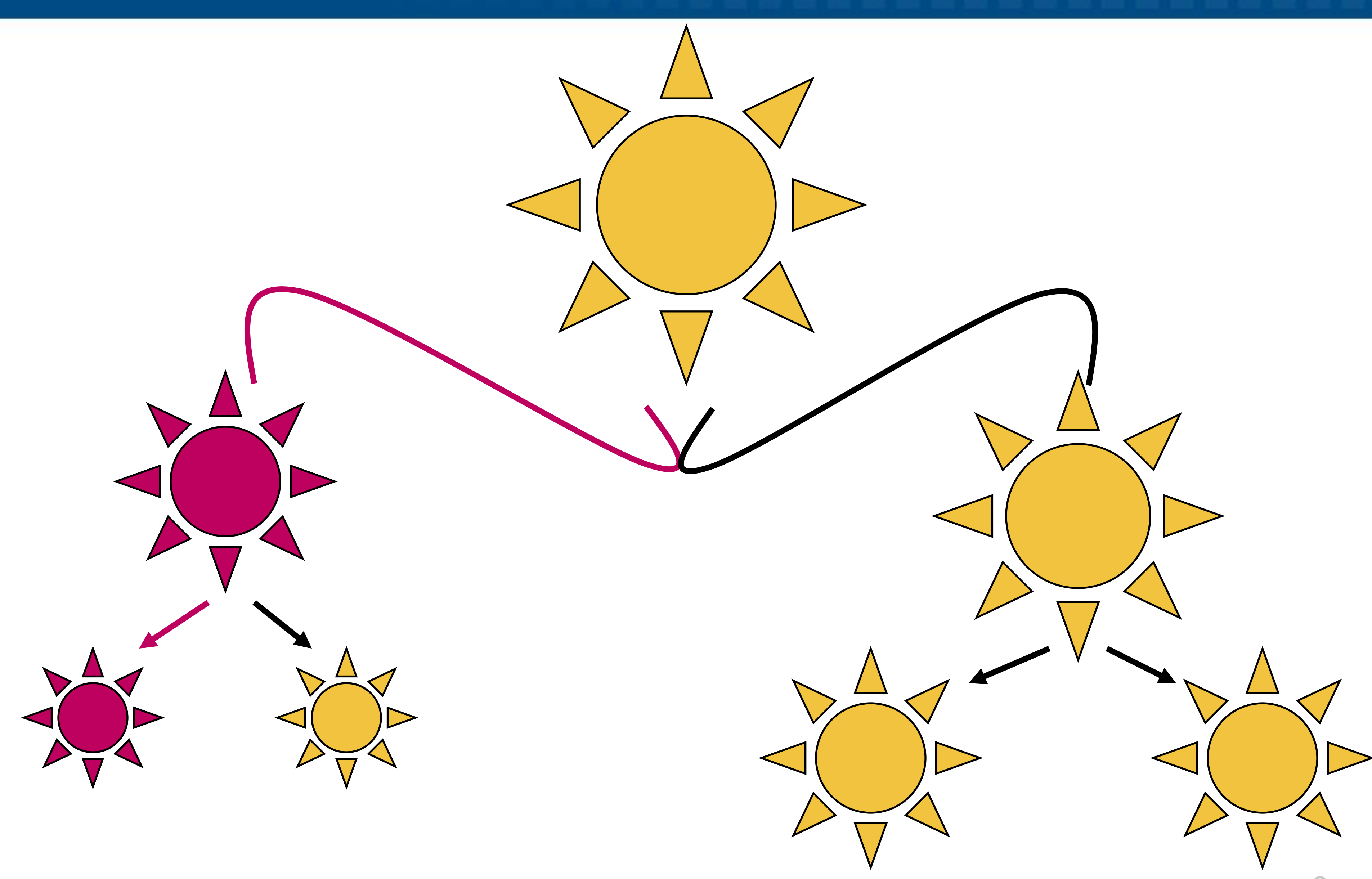
Chương 6

PHƯƠNG PHÁP THIẾT KẾ THUẬT TOÁN – CHIA ĐỂ TRỊ –

Nội dung

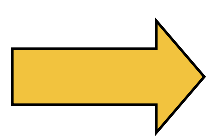
- Giới thiệu
- Phương pháp
- Sơ đồ cài đặt
- Các ví dụ

Hình ảnh



Giới thiệu

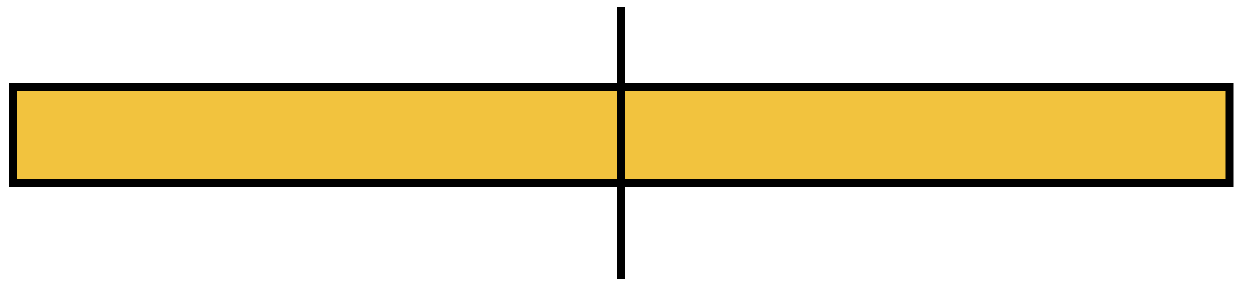
- Chia để trị là phương pháp thiết kế thuật toán từ trên xuống dưới (top – down) với ý tưởng:
  - Chia bài toán lớn thành những bài toán nhỏ hơn có dạng giống bài toán ban đầu
  - Các bài toán nhỏ hơn được chia thành những bài toán nhỏ hơn nữa



với hy vọng rằng các bài toán nhỏ dễ giải hơn

Phương pháp

- Phương pháp Chia để trị gồm 3 bước:
  - Bước 1 [Divide] – Chia bài toán thành các phần.
  - Bước 2 [Solve] – Giải quyết các phần
  - Bước 3 [Combine] – Kết hợp các lời giải của các phần thành lời giải của bài toán



Phương pháp

- Nhận xét quan trọng:
  - Các bài toán con (các phần) nhận được trong quá trình phân chia sẽ cùng dạng với bài toán ban đầu, chỉ khác nhau về kích thước
  - Có thể có một số bài toán con không cùng dạng với bài toán lớn
  - Các bài toán con Không được giao nhau

Sơ đồ cài đặt

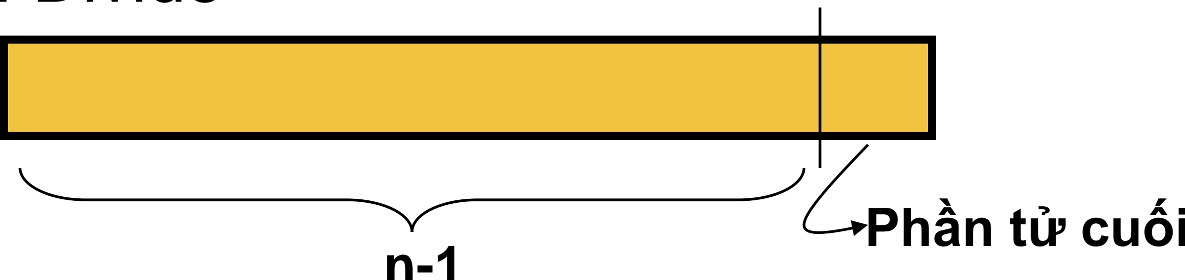
- Cài đặt bằng phương pháp Đệ qui

```
void DivideConquer(A, x)
{
      if (A du nho) Solve(A)
      else
      {
            - Phan chia A thanh A0, A1, …, An-1
            - for (i=0; i<n; i++)
                  DivideConquer(Ai, xi)
            - Ket hop cac nghiem xi de duoc nghiem x
      }
}
```

Các ví dụ

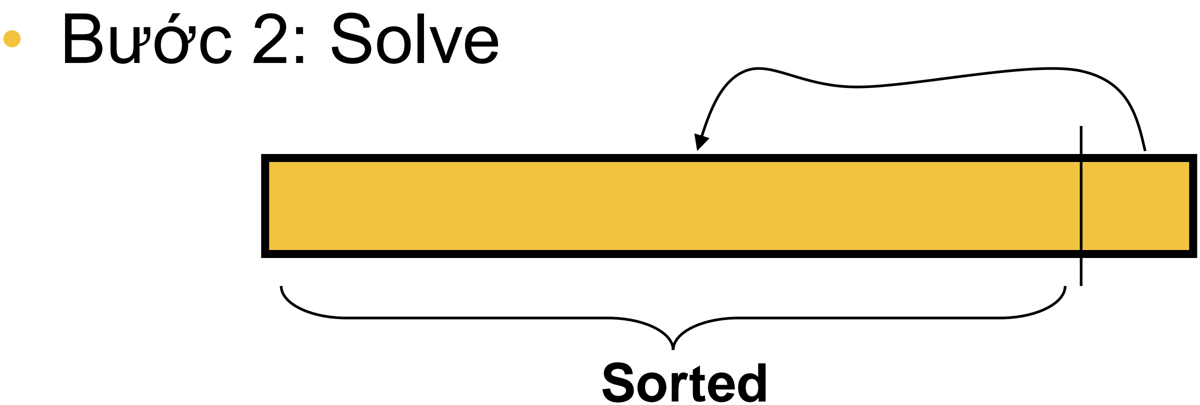
- Ví dụ 1 [Sorting 1]: Cho dãy a_1, a_2, …, a_n. Hãy xây dựng thuật toán sắp xếp dãy trên tăng dần.

  - Bước 1: Divide

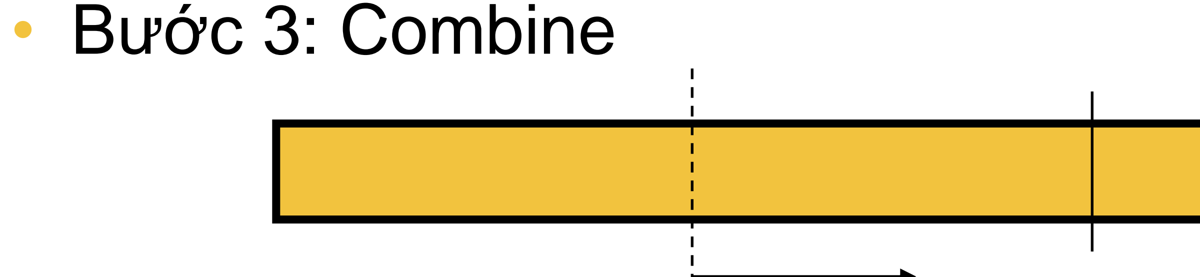


Các ví dụ

- Bước 2: Solve



- Bước 3: Combine



Các ví dụ

- Thuật toán **Insertion sort** **1** [Đệ quy – từ trên xuống]: Giả sử cần sắp xếp dãy a_1, a_2, …, a_i
  - Bước 1: Sắp xếp dãy a_1, a_2, a_{i-1} tăng dần
  - Bước 2: Tìm vị trí thích hợp trong dãy để chèn a_i vào sao cho a_1, a_2, …, a_i tăng dần

Các ví dụ

```
void InsertionSort1(int a[], int i)
{











}
```

Các ví dụ

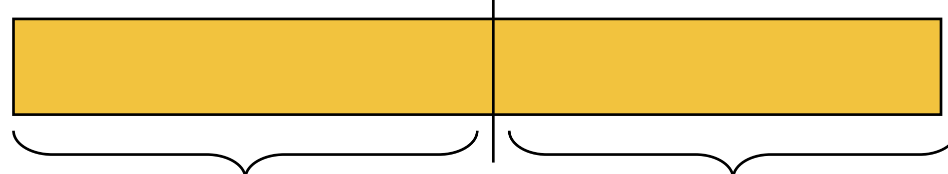
- Thuật toán **Insertion sort 2** [Vòng lặp – từ dưới lên]
  - a_1 đã được sắp xếp
  - Giả sử dãy a_1, a_2, …, a_{i-1} đã tăng dần
  - Tìm vị trí thích hợp trong dãy để chèn a_i vào sao cho a_1, a_2, …, a_i tăng dần

Các ví dụ

```
void InsertionSort2(int a[], int i)
{



}
```

Các ví dụ

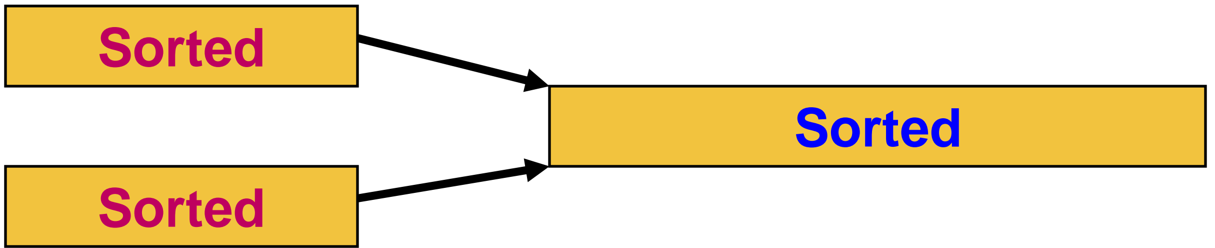
- Ví dụ 2 [Sorting 2]: Cho dãy a_1, a_2, …, a_n. Hãy xây dựng thuật toán sắp xếp dãy trên tăng dần.
  - Bước 1: Divide
    - Chia thành 2 phần "bằng nhau"



  - Bước 2: Solve
    - Sắp xếp mỗi phần tăng dần

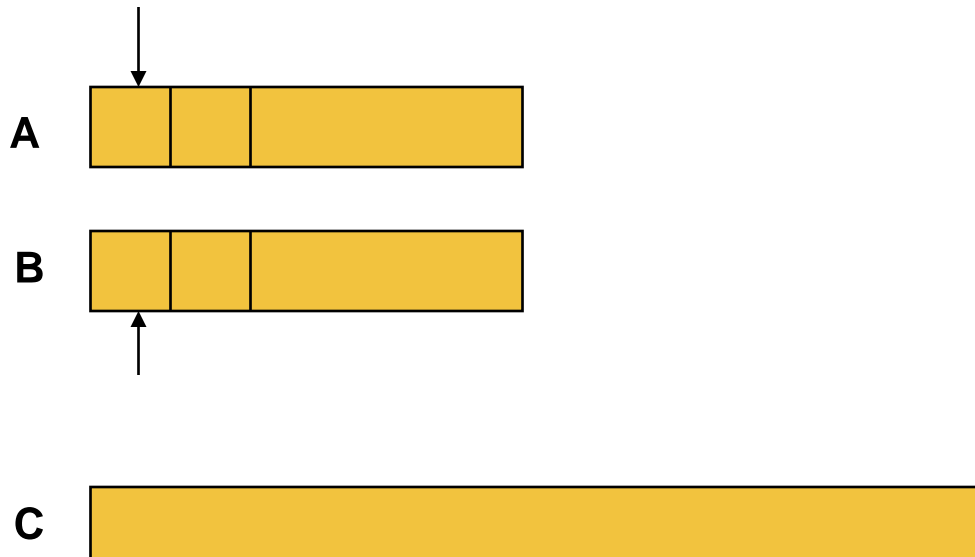
Các ví dụ

- Bước 3: Combine
  - Kết hợp 2 phần đã sắp xếp thành 1 phần được sắp xếp



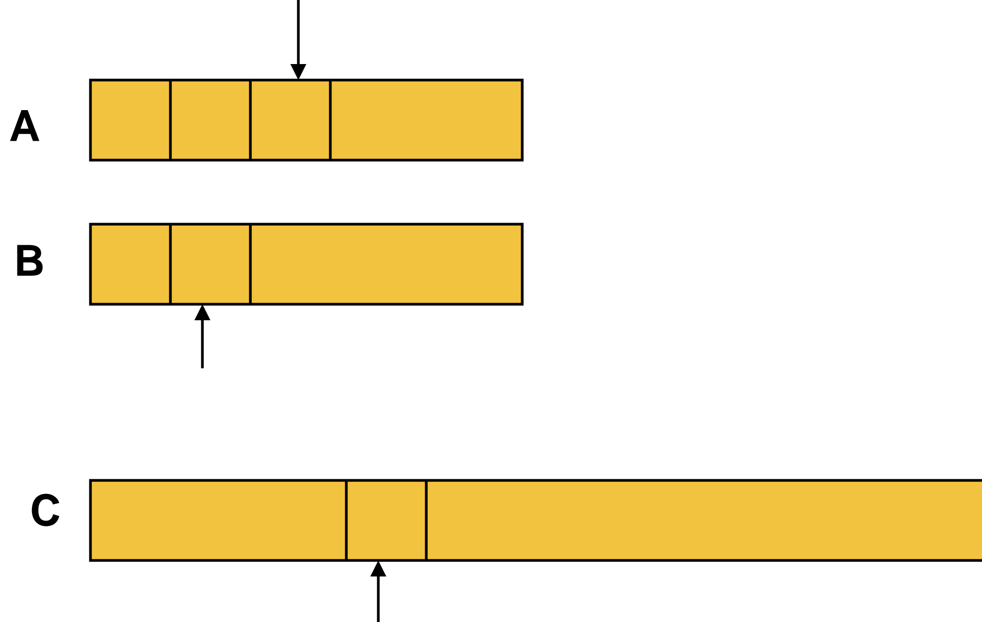
Các ví dụ

- Phương pháp trộn 2 dãy đã được sắp xếp thành 1 dãy được sắp xếp



Các ví dụ

- Phương pháp trộn 2 dãy đã được sắp xếp thành 1 dãy được sắp xếp



Các ví dụ

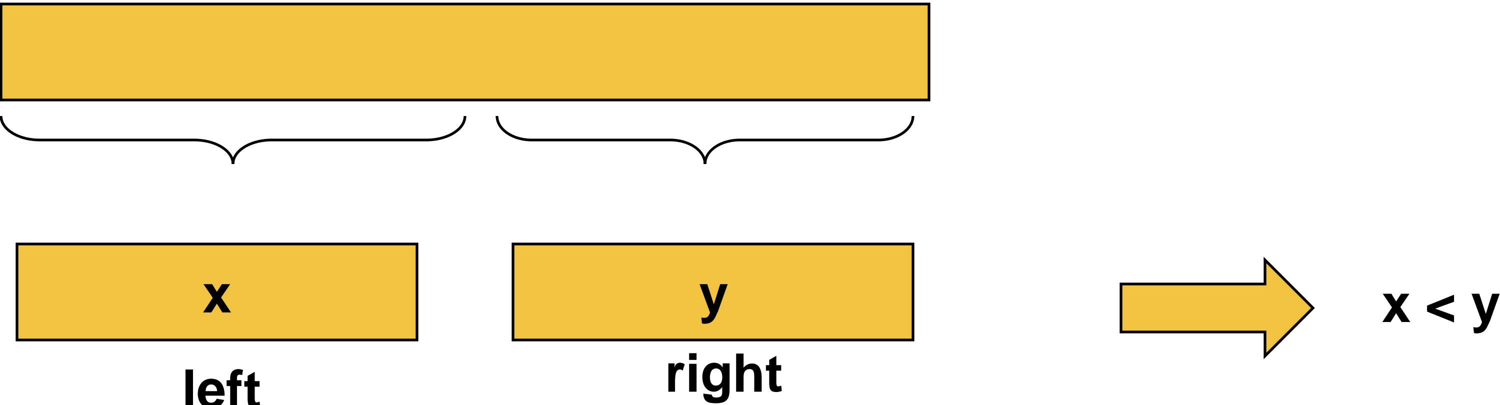
- Thuật toán **Merge sort**
  - Bước 1: Tính k = n div 2
  - Bước 2: Sắp xếp a[1…k]
  - Bước 3: Sắp xếp a[(k+1)…n]
  - Bước 4: Trộn 2 dãy đã sắp xếp a[1…k] và a[(k+1)…n] thành dãy a[1…n] được sắp xếp

Các ví dụ

```
void MergeSort(int a[], int i, int j)
{




}
```

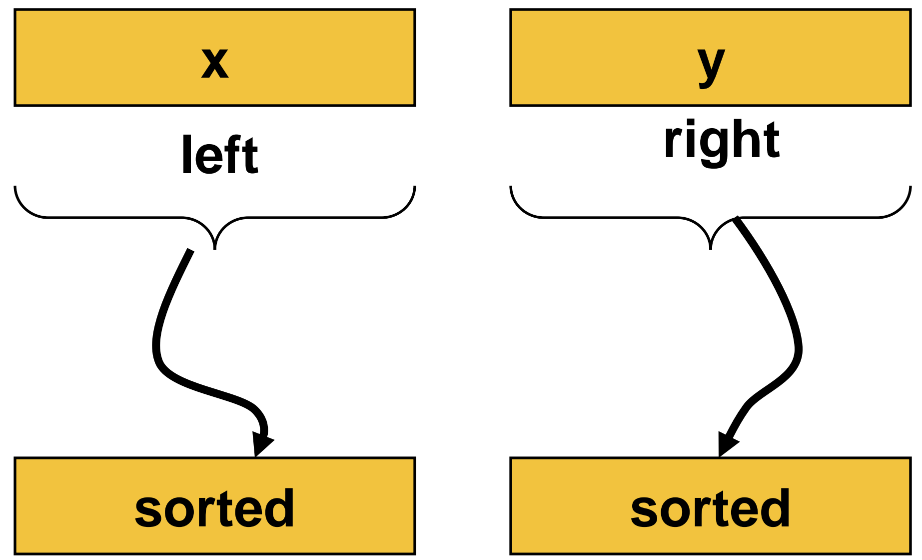
Các ví dụ

- Ví dụ 3 [Sorting 3]: Cho dãy $a_1, a_2, \ldots, a_n$. Hãy xây dựng thuật toán sắp xếp dãy trên tăng dần.
  - Bước 1: Divide
    - Chia thành 2 phần



Các ví dụ

- Bước 2: Solve
  - Sắp xếp phần bên trái và phần bên phải



- Bước 3: Combine
  - Đặt 2 phần kề nhau

Các ví dụ

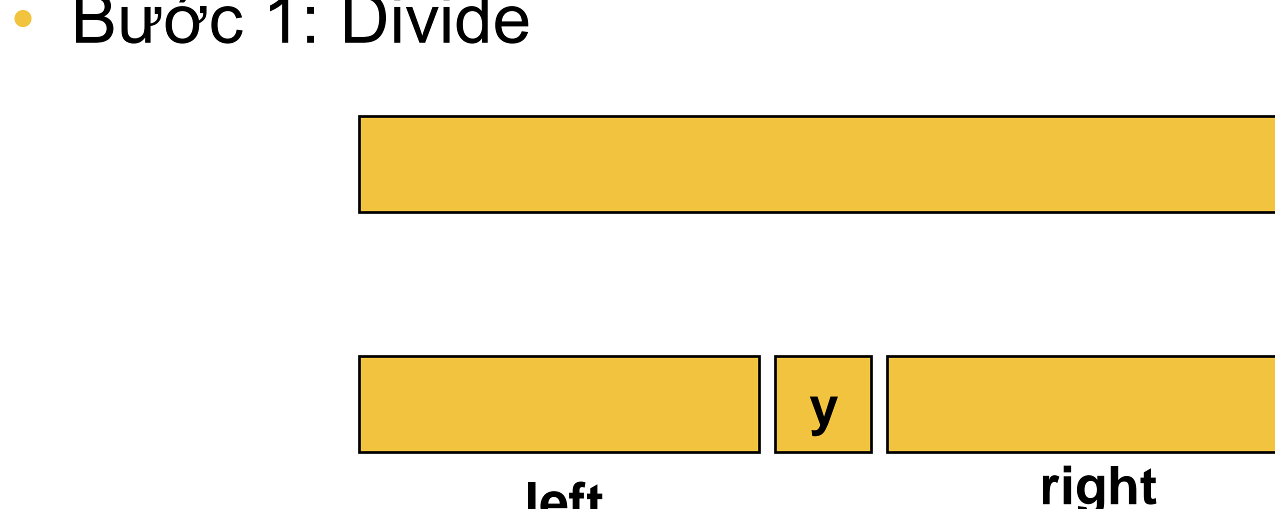
- Thuật toán **Quick sort**
  - Bước 1: Chọn phần tử trung tâm p
  - Bước 2: Chia làm 2 phần
    - Phần bên trái: Gồm những phần tử nhỏ hơn p
    - Phần bên phải: Gồm những phần tử lớn hơn hay bằng p
  - Bước 3: Sắp xếp phần bên trái và phần bên phải một cách đệ quy

Các ví dụ

```
void QuickSort(int a[], int left, int right)
{

}
```

Các ví dụ

- Ví dụ 4: [Tìm kiếm nhị phân]
  - Bài toán: Cho dãy đã được sắp xếp tăng. Hãy kiểm tra xem x có trong dãy hay không
  - Bước 1: Divide



Các ví dụ

- Bước 2: Solve
  - Kiểm tra x với y:
    - x = y → Tìm thấy
    - x < y: Tìm bên left
    - x > y: Tìm bên right
- Bước 3: Combine
  - Không làm gì cả

Các ví dụ

- Thuật toán Binary search: Tìm kiếm x có trong dãy a[l… r]
  - Bước 1: Nếu l>r thì không tìm thấy
  - Bước 2: Tính m = (l+r)/2
  - Bước 3:
    - Nếu x = a[m] thì tìm thấy
    - Nếu x < a[m] thì tìm bên a[l…m-1]
    - Nếu x > a[m] thì tìm bên a[m+1…r]

Các ví dụ

```
int BinarySearch(int a[], int left, int right, int x)
{
}
```

Tóm tắt chương 6

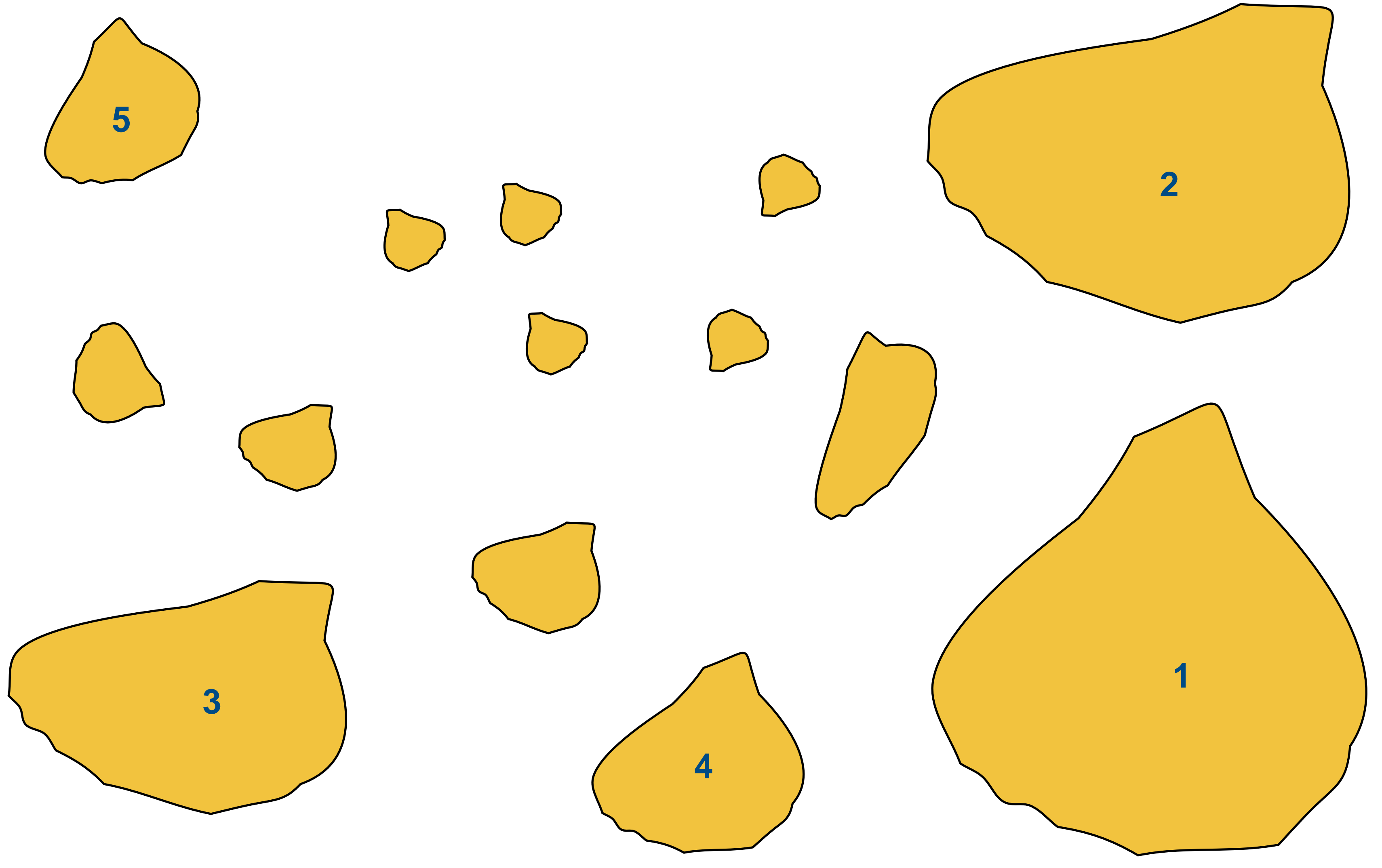
Chương 7

PHƯƠNG PHÁP THIẾT KẾ THUẬT TOÁN
– THAM LAM –

Nội dung

- Giới thiệu
- Phương pháp
- Sơ đồ cài đặt
- Các ví dụ
- Ưu điểm và khuyết điểm

Hình ảnh



Giới thiệu

- Định nghĩa [Tham lam – Greedy]: Tham lam là một phương pháp thiết kế thuật toán để tìm nghiệm của bài toán tối ưu bằng cách xây dựng nghiệm dần dần từng bước. Tại mỗi bước:
  - Chúng ta luôn luôn chọn giá trị tốt nhất tại thời điểm đó mà không quan tâm đến tương lai (tối ưu cục bộ)
  - Chúng ta hy vọng việc chọn các tối ưu cục bộ tại mỗi bước sẽ cho tối ưu toàn cục

Phương pháp

- Phát biểu bài toán: Giả sử bài toán yêu cầu tìm phương án $X=(x_1, x_2, \ldots, x_n)$, trong đó
  - x_i được chọn ra từ tập D_i.
  - f(X) là hàm đánh giá sự tốt nhất của phương án X (f là hàm mục tiêu hay hàm chi phí)

Phương pháp

- Phương pháp Tham lam
  - Phương pháp Tham lam xây dựng dần nghiệm X của bài toán:
    - Ban đầu X=( )
    - Giả sử đã xây dựng được (k-1) thành phần của nghiệm $(x_1, x_2, \ldots, x_{k-1})$
    - Bây giờ ta mở rộng nghiệm thành $(x_1, x_2, \ldots, x_{k-1}, x_k)$ bằng cách chọn x_k là giá trị tốt nhất trong tập D_k

Phương pháp

- Phương pháp Tham lam
  - Thông thường tập D được sắp theo một trật tự tăng dần hay giảm dần theo tiêu chí nào đó từ đó giúp việc chọn giá trị tốt nhất cho x_i sẽ dễ dàng hơn
    - Bước 1 [Sắp xếp]: Sắp xếp dữ liệu D tăng dần hay giảm dần theo tiêu chí nào đó
    - Bước 2 [Chọn giá trị tốt nhất]: Với mỗi thành phần x_i. Ta tìm giá trị tốt nhất trong dữ liệu đã được sắp xếp trong bước 1 và thỏa điều kiện của bài toán để gán cho x_i

Sơ đồ cài đặt

- Sơ đồ 1:

```
void Greedy1()
{
        X=();
        for (i=1; i<=n; i++)
        {
                Xác định Di;
                xi = SelectBest(Di);
        }
}
```

Sơ đồ cài đặt

- Sơ đồ 2:

```
void Greedy2()
{
      Sort(D);
      for (i=1; i<=n; i++)
      {
            - Chọn v là giá trị tốt nhất trong D và
                  thỏa điều kiện bài toán
            - xi = v;
            - Bỏ v khỏi D
      }
}
```

Chú ý

- Một ý tưởng đối nghịch với phương pháp "tham lam" là ý tưởng "trông rộng":
  - Gom nhỏ thành to
  - Năng nhặt chặt bị

Chú ý

- Để giải bài toán bằng phương pháp tham lam, chúng ta cần:
  - Xác định các tập giá trị D_i
  - Hàm mục tiêu f
  - Hàm chọn SelectBest để chọn giá trị cho x_i

Các ví dụ: {1} Bài toán thu nhạc

- Ví dụ 1 [Bài toán thu nhạc]
  Một băng đĩa có thể thu được các bài hát với tổng thời lượng là T. Có N bài hát, bài thứ i có thời lượng là h_i khi lưu trên đĩa (i=1, 2, …, N)

- Yêu cầu: Hãy chọn một cách thu các bài hát sao cho mỗi bài chỉ thu một lần và tổng số bài thu được trên băng là nhiều nhất

Các ví dụ: {1} Bài toán thu nhạc

- Biểu diễn lời giải của bài toán là 1 vector độ dài k: $X=(x_1, x_2, \ldots, x_k)$. Trong đó $x_i = 1, 2, \ldots, n$
  - [illegible]
  - $f(X)=|X| \rightarrow \max$

- Bài toán: Tìm vector X

- Thuật toán tham lam:
  - Chọn bài có thời lượng nhỏ thu trước, bài có thời lượng lớn thu sau nếu còn chỗ

Các ví dụ: {1} Bài toán thu nhạc

```
void ThuNhac_Greedy()
{









}
```

Các ví dụ: {2} Bài toán cái túi

- Ví dụ 2 [Bài toán cái túi – 0-1 Knapsack problem] Cho n loại đồ vật được đánh số từ 1 đến n, đồ vật thứ i có
  - v_i – giá trị của đồ vật i
  - w_i – trọng lượng đồ vật i
- Yêu cầu: Tìm một số đồ vật để bỏ vào túi sao cho tổng trọng lượng các đồ vật bỏ vào túi không vượt quá W và tổng giá trị của các đồ vật là lớn nhất.

Các ví dụ: {2} Bài toán cái túi

- Biểu diễn lời giải của bài toán là 1 vector nhị phân độ dài n: $X=(x_1, x_2, \ldots, x_n)$. ($x_i$ [illegible], 1})
  - $x_i=1$: Chọn đồ vật i
  - $x_i=0$: Không chọn đồ vật i
  - Trọng lượng của nghiệm thành phần: x_i*w_i
  - Giá trị của nghiệm thành phần: x_i*v_i

- Bài toán: Tìm vector X

Các ví dụ: {2} Bài toán cái túi

- Thuật toán tham lam 1:
  - Bước 1: Sắp xếp các đồ vật có giá trị giảm dần
  - Bước 2:
    - TrongLuong=0
    - $x_i=0$
  - Bước 3: Xét tuần tự các đồ vật từ trái sang phải. Với đồ vật thứ i:
    - Nếu TrongLuong + $w_i < W$ thì
      - Chọn đồ vật i: $x_i=1$
      - TrongLuong = TrongLuong + w_i

Các ví dụ: {2} Bài toán cái túi

- Thuật toán tham lam 2:
  - Sắp xếp các đồ vật có giá trị tăng dần
- Thuật toán tham lam 3:
  - Sắp xếp các đồ vật có giá trị trên 1 đơn vị trọng lượng (v_i/w_i) giảm dần
- Thuật toán tham lam 4:

Các ví dụ: {2} Bài toán cái túi

```
void KnapSack_Greedy()
{













}
```

Các ví dụ: {3} Bài toán người du lịch

- Ví dụ 3 [Bài toán người du lịch – Traveling Salesman Problem – TSP]
  Cho n thành phố được đánh số từ 1 đến n và khoảng cách giữa thành phố i và thành phố j được cho bởi c_{ij} (chú ý: $c_{ij}=c_{ji}$)

  Yêu cầu: Tìm một hành trình ngắn nhất cho phép viếng thăm n thành phố, mỗi thành phố viếng thăm đúng 1 lần và quay về thành phố ban đầu.

Các ví dụ: {3} Bài toán người du lịch

- Biểu diễn lời giải của bài toán là 1 vector độ dài n: $X=(x_1, x_2, \ldots, x_n)$. ($x_1 = 1$). Trong đó $(x_1, x_2, \ldots, x_n)$ là một hoán vị của $(1, 2, \ldots, n)$

- Bài toán: Tìm vector X

Các ví dụ: {3} Bài toán người du lịch

- Thuật toán tham lam:
  - Ý tưởng: Xuất phát từ thành phố số 1, tại mỗi bước ta sẽ chọn thành phố tiếp theo là thành phố chưa viếng thăm và có khoảng cách từ thành phố hiện tại đến thành phố đó là nhỏ nhất

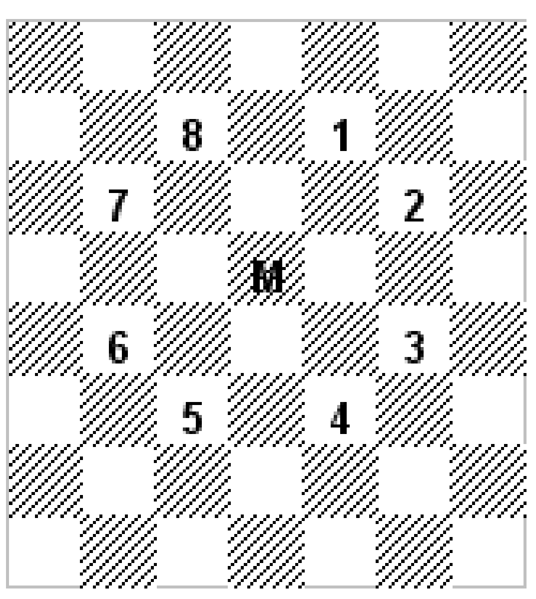
  - Bước 1: $x_1=1$; $x_n=1$
  - Bước 2: Chọn x_i là thành phố chưa đi qua và có khoảng cách đến x_{i-1} là nhỏ nhất.

Các ví dụ: {3} Bài toán người du lịch

```
void TSP_Greedy()
{
















}
```

Các ví dụ: {4} Bài toán mã đi tuần

- Ví dụ 4 [Bài toán mã đi tuần]
  Trên bàn cờ quốc tế có một con mã nằm tại một ô nào đó. Hãy chỉ ra 1 cách di chuyển con mã trên bàn cờ theo luật đi con mã sao cho mỗi ô trên bàn cờ, con mã nhảy đến đúng một lần.



Các ví dụ: {4} Bài toán mã đi tuần

- Thuật toán tham lam:
  - Ở gần biên sẽ có ít nước đi hơn các ô bên trong
  - Ý tưởng: Ưu tiên đi ra biên để đi những ô có ít nước đi nhất rồi mới đi đến những ô bên trong

Các ví dụ: {4} Bài toán mã đi tuần

```
void Horse_Greedy()
{



}
```

Ưu điểm và khuyết điểm

- Ưu điểm
  - Tìm được các nghiệm gần tối ưu
  - Thời gian thực thi nhanh hơn các phương pháp tối ưu, quay lui
- Khuyết điểm
  - Nghiệm tìm được có thể không tốt nhất

Tóm tắt chương 7

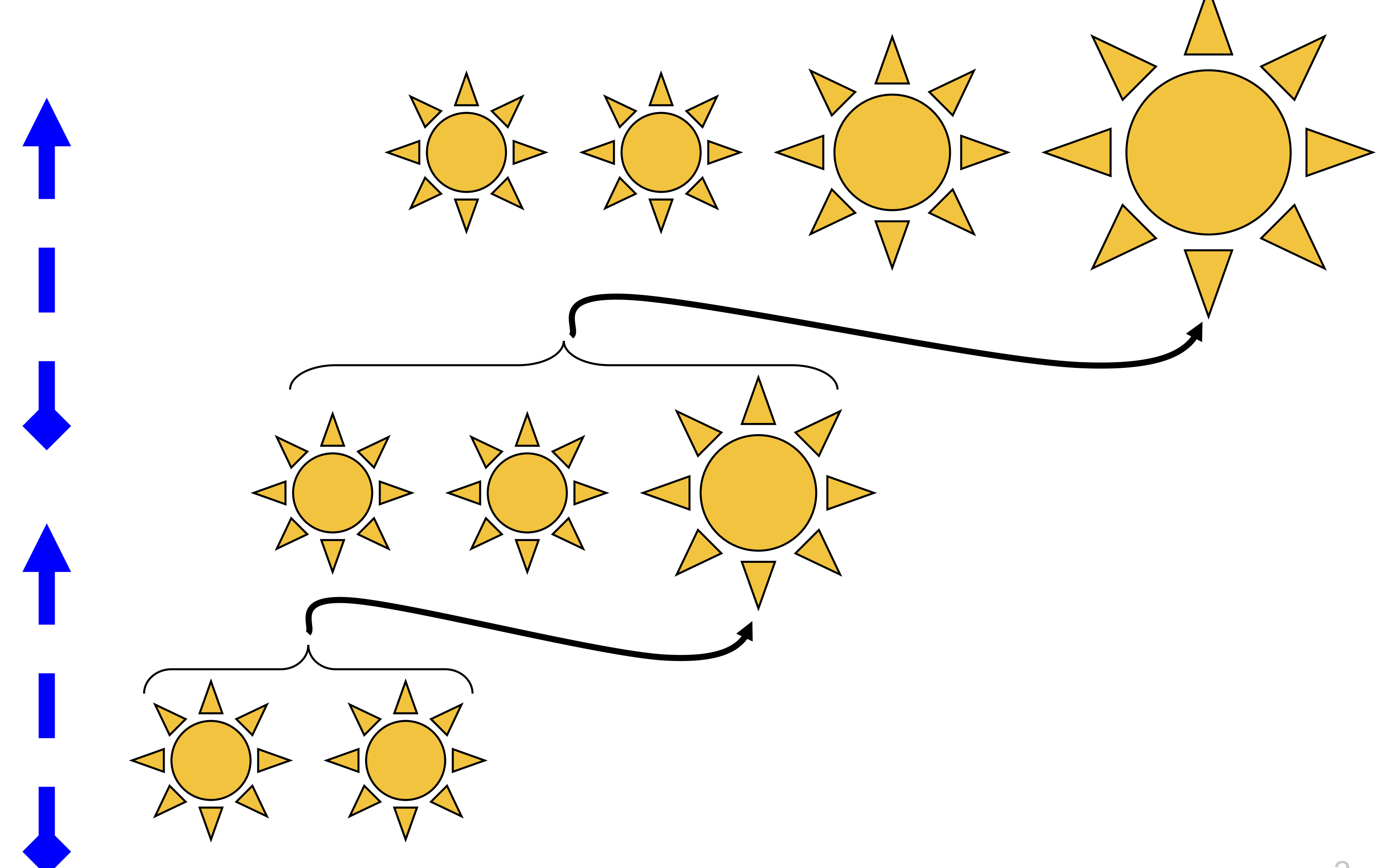
Chương 8

PHƯƠNG PHÁP THIẾT KẾ THUẬT TOÁN
– QUY HOẠCH ĐỘNG –

Nội dung

- Giới thiệu
- Quy hoạch động và Chia để trị
- Quy hoạch động và Bài toán tối ưu
- Nguyên lý tối ưu của Bellman
- Sơ đồ cài đặt
- Các ví dụ

Hình ảnh



Giới thiệu

- Quy hoạch động – Dynamic Programming do nhà toán học người Mĩ Richard Bellman (1920 – 1984) phát minh vào năm 1957

- Quy hoạch động – Dynamic Programming là phương pháp để giải quyết một lớp lớn các bài toán tối ưu thỏa theo nguyên lý tối ưu Bellman

Giới thiệu

- Dựa trên phương pháp Quy hoạch động, nhiều thuật toán nổi tiếng đã ra đời: Một số thuật toán nổi tiếng dựa trên phương pháp Quy hoạch động
  - Thuật toán Dijkstra
  - Thuật toán Ford – Bellman
  - Thuật toán Floyd
  - Thuật toán Viterbi
  - Thuật toán huấn luyện Adaptive Critic
  - Thuật toán Cocke – Younger – Kasami
  - …

Quy hoạch động và Chia để trị

Bài toán con trùng lắp
(Overlapping subproblems)

Phương pháp

- Phương pháp Quy hoạch động gần giống với phương pháp Chia để trị.
- Cả hai phương pháp dùng để giải quyết bài toán bằng cách kết hợp các lời giải của các bài toán con.

Phương pháp

- Phương pháp Chia để trị: Là phương pháp từ trên xuống dưới (top – down) với ý tưởng:
  - [Divide] Chia bài toán lớn thành những bài toán nhỏ hơn và độc lập nhau
  - [Solve] Giải quyết các bài toán nhỏ
  - [Combine] Kết hợp các lời giải bài toán nhỏ để hình thành lời giải bài toán lớn

Phương pháp

- Hạn chế của phương pháp Chia để trị:
  - Khi dùng phương pháp chia để trị để chia 1 bài toán lớn thành các bài toán con, các bài toán con lại chia nhỏ thành nhiều bài toán con nhỏ hơn nữa, …
    ➔ Đôi khi một bài toán con được yêu cầu giải nhiều lần
    ➔ Chương trình chạy chậm

Phương pháp

- Phương pháp Quy hoạch động: Là phương pháp giải quyết bài toán bằng cách:
  - [Solve & Restore] Giải quyết các bài toán nhỏ nhất, rồi lưu kết quả lại
  - [Combine & Restore] Kết hợp các lời giải của bài toán nhỏ để hình thành lời giải của bài toán lớn, rồi lưu kết quả lại

2 Tiếp cận cài đặt Quy hoạch động

- Tiếp cận từ Dưới lên (Bottom Up):
  - Toàn bộ các bài toán con nhỏ nhất cần giải sẽ được giải trước
  - Sử dụng các kết quả để tìm nghiệm của bài toán lớn hơn
  - …
  - Quá trình tiếp tục cho đến khi bài toán cuối được giải

2 Tiếp cận cài đặt Quy hoạch động

- Sơ đồ cài đặt

```
void SolveSmallProblems()
{
}

void SolveSubProblems()
{
}

void Trace()
{
}

void DynamicProgramming()
{
      SolveSmallProvlems();
      SolveSubProblems();
      //Trace();
      …
}
```

2 Tiếp cận cài đặt Quy hoạch động

- Ưu điểm của tiếp cận Bottom – Up
  - Tốn ít bộ nhớ

- Khuyết điểm của tiếp cận Bottom – Up
  - Cài đặt dài hơn tiếp cận Top – Down
  - Vì để tiết kiệm bộ nhớ nên bài toán con nào dùng xong mà không dùng nữa thì bỏ đi → Sau khi giải xong sẽ không xem được trình tự quá trình giải (không lưu lại lịch sử)

2 Tiếp cận cài đặt Quy hoạch động

- Tiếp cận từ trên xuống (Top Down) – Dùng đệ qui có nhớ (Memoization)
  - [Divide] Chia bài toán thành các bài toán con
  - [Solve]
    - Trước khi giải bài toán con, chúng kiểm tra xem bài toán này đã được giải trước đó chưa.
      - Nếu đã giải thì lấy lời giải trong bảng ra
      - Nếu chưa giải thì giải
    - Sau khi có lời giả thì chúng lưu kết quả lại vào bảng
  - [Combine] Kết hợp các lời giải của các bài toán con thành lời giải của bài toán

2 Tiếp cận cài đặt Quy hoạch động

- Sơ đồ cài đặt

```
void DynamicProgramming(A, x)
{
      if (A đã giải quyết)
            x = LoiGiai(A); // Lấy lời giải từ bộ nhớ

      if (A du nho)
            LoiGiai(A) = Solve(A); //lưu lời giải bài
                                   //toán A vào bộ nhớ
      else
      {
            - Phan chia A thanh A0, A1, …, An-1
            - for (i=0; i<n; i++)
                  DynamicProgramming(Ai, xi)
            - Ket hop cac nghiem xi de duoc nghiem x
            - LoiGiai(A) = x;
      }
}
```

2 Tiếp cận cài đặt Quy hoạch động

- Ưu điểm của tiếp cận Top – Down
  - Cài đặt ngắn gọn
  - Có thể quan sát các bài toán con cần giải
- Khuyết điểm của tiếp cận Top – Down
  - Tốn nhiều bộ nhớ vì phải lưu toàn bộ các bài toán con đã giải vì không biết bài toán con đó còn dùng nữa hay không

Ví dụ

- Tính số Fibonacci thứ n

- Hai tiếp cận bằng Quy hoạch động
  - Dùng tiếp cận từ trên xuống
  - Dùng tiếp cận từ dưới lên

Ví dụ

```
void QHD_TopDown(int n)
{

}
```

Ví dụ

```
void QHD_BottomUp(int n)
{

}
```

Quy hoạch động và
Bài toán tối ưu

Phương pháp

- Định nghĩa Quy hoạch động: Quy hoạch động là phương pháp giải quyết các bài toán tối ưu bằng cách tạo ra một chuỗi các quyết định xác định. Với mỗi quyết định, các bài toán con được giải quyết theo cùng cách sao cho lời giải tối ưu của bài toán ban đầu có thể được tìm thấy từ các lời giải tối ưu của các bài toán con.

Phương pháp

- Phương pháp Quy hoạch động dựa trên nguyên tắc tối ưu của Bellman
- Nguyên lý tối ưu của Bellman:
    - Trong một quá trình quyết định có nhiều giai đoạn, Lời giải tối ưu có thuộc tính:
        - Dù trạng thái ban đầu và các quyết định ban đầu như thế nào đi nữa thì những quyết định còn lại phải tạo thành lời giải tối ưu mà không phụ thuộc vào trạng thái được sinh ra từ những quyết định ban đầu

Phương pháp

- Bellman's Principle of Optimality
  - An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.

Phương pháp

- Chúng ta gọi:
  - Hàm mục tiêu là f
  - Giá trị tối ưu là f*
  - Các quyết định là d_1, d_2, …

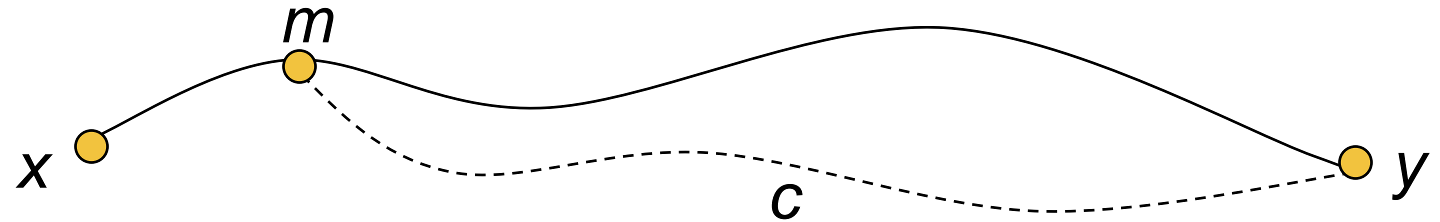
$$f^{*} \;\text{[illegible]}_{d_1,d_2,\ldots,d_n}\{f(d_1,d_2,\ldots,d_n)\}$$

$$\text{[illegible]}t_{d_2}\{\ldots\{opt_{d_n}\{f(d_1,d_2,\ldots,d_n)\}\}\ldots\}\}$$

Phương pháp

- Nguyên lý tối ưu của Bellman:
  - Nói cách khác ngắn gọn hơn:
    - Lời giải tối ưu cho bài toán P phải chứa lời giải tối ưu của các bài toán con của P (Lời giải tối ưu có lời giải con tối ưu)
  - Nguyên lý trên phù hợp với nhận xét rằng: Nếu lời giải mà có lời giải con không tối ưu thì khi thay lời giải con đó bằng lời giải con tối ưu sẽ cho lời giải tối ưu hơn lời giải ban đầu

Phương pháp



- Giả thiết: Nếu xmy là đường tối ưu từ x đến y. Đường đi này qua m thì my là đường đi tối ưu
- Chứng minh: Giả sử không đúng. Thế thì có một đường khác là đường c là đường đi tối ưu từ m đến y. Nghĩa là: Đường đi từ x đến m, rồi theo đường c đến y sẽ tối ưu hơn đường đi ban đầu.
- Đều này mâu thuẫn với giả thiết là xmy là đường đi tối ưu từ x đến y

Phương pháp

- Nhận xét:
  - Nguyên lý tối ưu của Bellman còn được gọi là thuộc tính cấu trúc con tối ưu (Optimal substructure)
  - Việc giải bài toán tối ưu sẽ đưa về giải bài toán con tối ưu cùng dạng
  - Để việc tính toán của phương pháp DP đạt hiệu quả thì các lời giải của các bài toán con được sử dụng nhiều lần chỉ nên được giải 1 lần rồi lưu trữ lại để sử dụng lại sau này

Phương pháp

- Các bước giải bài toán tối ưu theo Quy hoạch động:
  - Bước 1 [Xác định cấu trúc con tối ưu]:
    - Chọn số tham số cho Hàm mục tiêu f
    - (Hàm mục tiêu dùng để biểu diễn cấu trúc con của bài toán)
    - Số tham số của hàm mục tiêu f phụ thuộc vào
      - Số đại lượng tham gia vào bài toán
      - Cấu trúc con tối ưu của từng bài toán tối ưu cụ thể

Phương pháp

- Bước 2 [Tính toán Trường hợp Cơ bản]:
  – Tính hàm mục tiêu f với các giá trị đơn giản nhất để biết hướng xây dựng các giá trị của hàm f

- Bước 3 [Tính toán Trường hợp Tổng quát]:
  – Tìm công thức cho hàm mục tiêu f
  – (Công thức/phương trình quy hoạch động)
  – (Công thức/phương trình Bellman)

Phương pháp

- Bước 4: [Tạo bảng phương án]
  - Tạo bảng lưu trữ các giá trị của hàm mục tiêu theo công thức đã tìm được trong bước 3
- Bước 5: [Truy vết]
  - Nếu bài toán yêu cầu chỉ ra tuần tự các quyết đã thực hiện, chúng ta cần truy vết từ quyết định cuối về quyết định ban đầu

Các ví dụ

- Ví dụ 1: [Dãy con tăng dài nhất]
  - Cho dãy số nguyên $A = a_1, a_2, \ldots, a_n$ ($n \leq 1000$, $-10000 \leq a_i \leq 10000$). Dãy con của A là một cách chọn ra trong A một số phần tử giữ nguyên thứ tự.
  - Yêu cầu: Hãy tìm một dãy con tăng của dãy A và có số phần tử nhiều nhất

Các ví dụ

Các ví dụ

- Ví dụ 2: [Đường đi lớn nhất]
  - Cho hình chữ nhật kích thước mxn (n,m ≤100), mỗi ô chứa một số nguyên. Có thể di chuyển từ ô (i, j) đến 1 trong 3 ô kề bên phải (i-1, j+1) (i, j+1) và (i+1, j+1) thuộc hình chữ nhật.
  - Yêu cầu: Hãy tìm một cách di chuyển từ một ô nào đó thuộc cột 1 đến 1 ô nào đó thuộc cột n sao cho tổng các số của các ô đi qua là lớn nhất.

Các ví dụ

Các ví dụ

- Ví dụ 3: [Dãy con chung dài nhất]
  - Cho 2 dãy số nguyên
    - $A = (a_1, a_2, \ldots, a_n)$.
    - $B=(b_1, b_2, \ldots, b_m)$
      ($n, m \le 100, -10000 \le a_i, b_j \le 10000$).
  - Yêu cầu: Hãy tìm một dãy C tăng và dài nhất, trong đó C là dãy con của A và C là dãy con của B

Các ví dụ

Tóm tắt chương 8

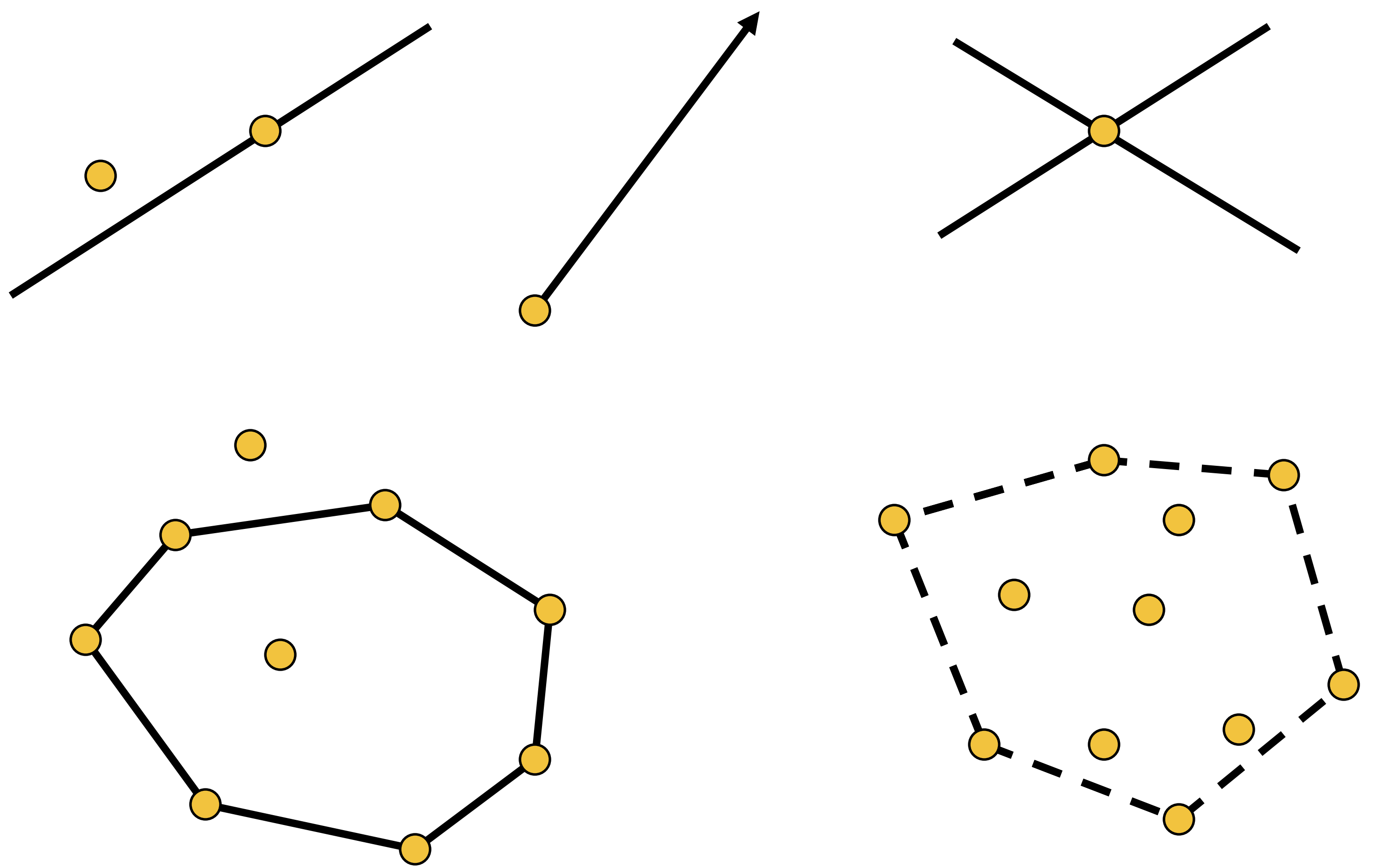
Chương 9

PHƯƠNG PHÁP THIẾT KẾ THUẬT TOÁN
– HÌNH HỌC –

Nội dung

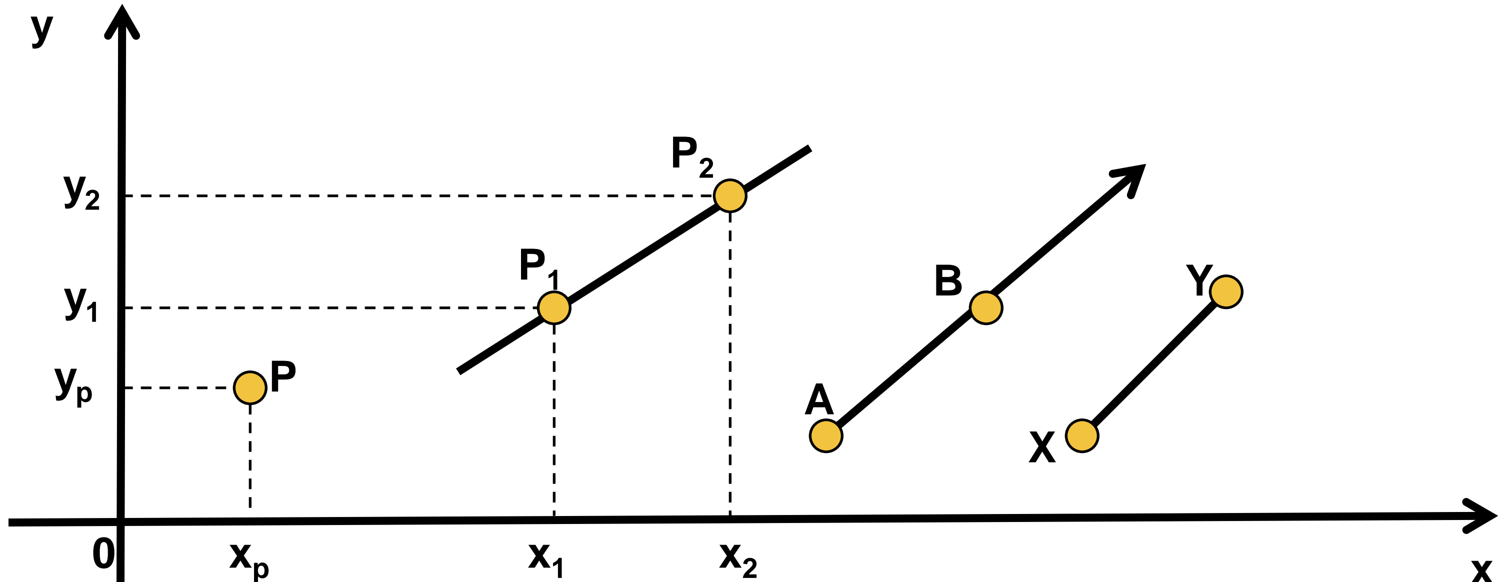
- Cấu trúc dữ liệu cơ bản
- Điểm và đoạn thẳng, đường thẳng và tia
- Giao điểm 2 đoạn thẳng, đường thẳng
- Đa giác
  - Điểm và đa giác
  - Đa giác lồi
  - Bao lồi

Hình ảnh



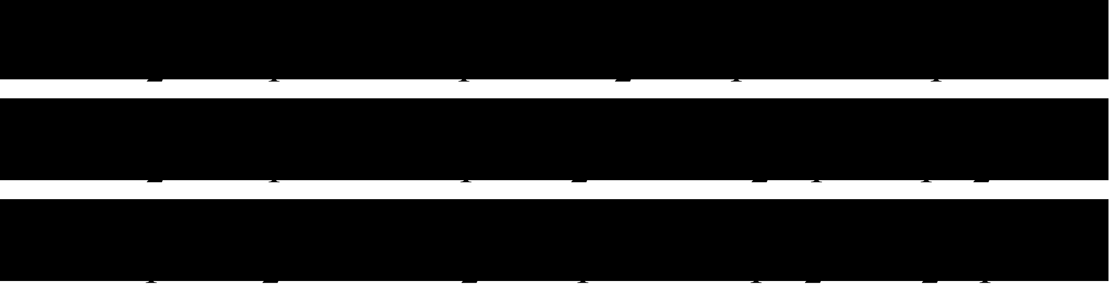
Cấu trúc dữ liệu cơ bản

- Một số cấu trúc dữ liệu hình học cơ bản
  - Điểm: $P(x_p, y_p)$
  - Đoạn thẳng: XY
  - Đường thẳng: Qua 2 điểm P_1, P_2
  - Tia: Tia AB



Cấu trúc dữ liệu cơ bản

- Phương trình của đường thẳng
  - Đường thẳng được xác định bởi 2 điểm $P_1(x_1, y_1)$, $P_2(x_2, y_2)$.

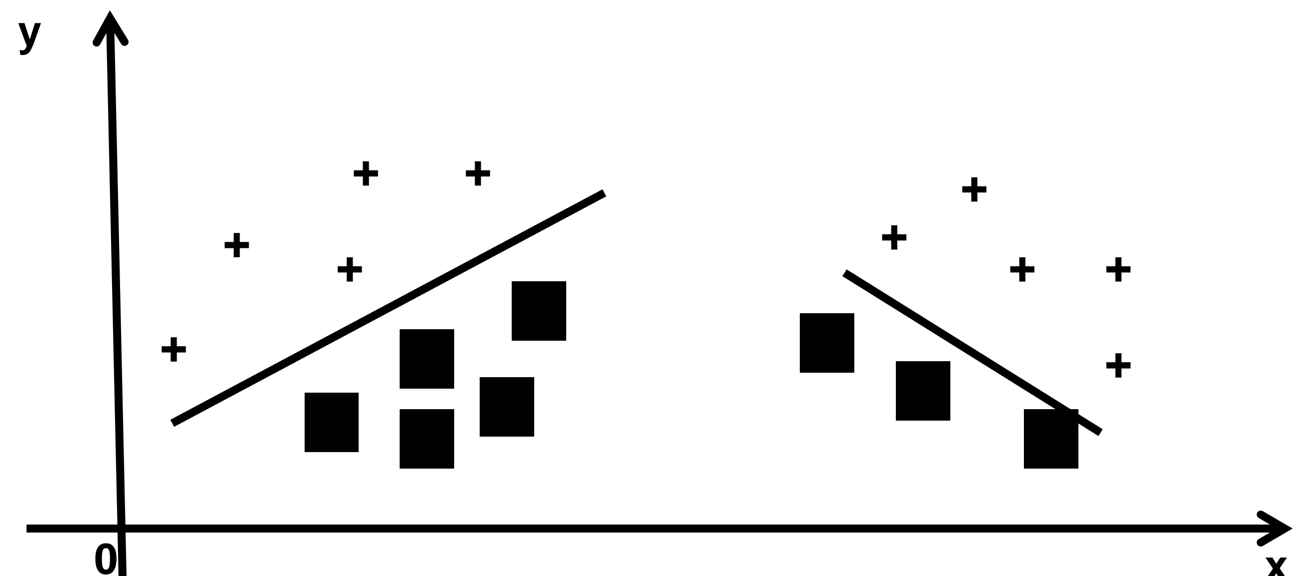


- Phương trình của đường thẳng
  - Dạng tổng quát

$$F(x, y)$$

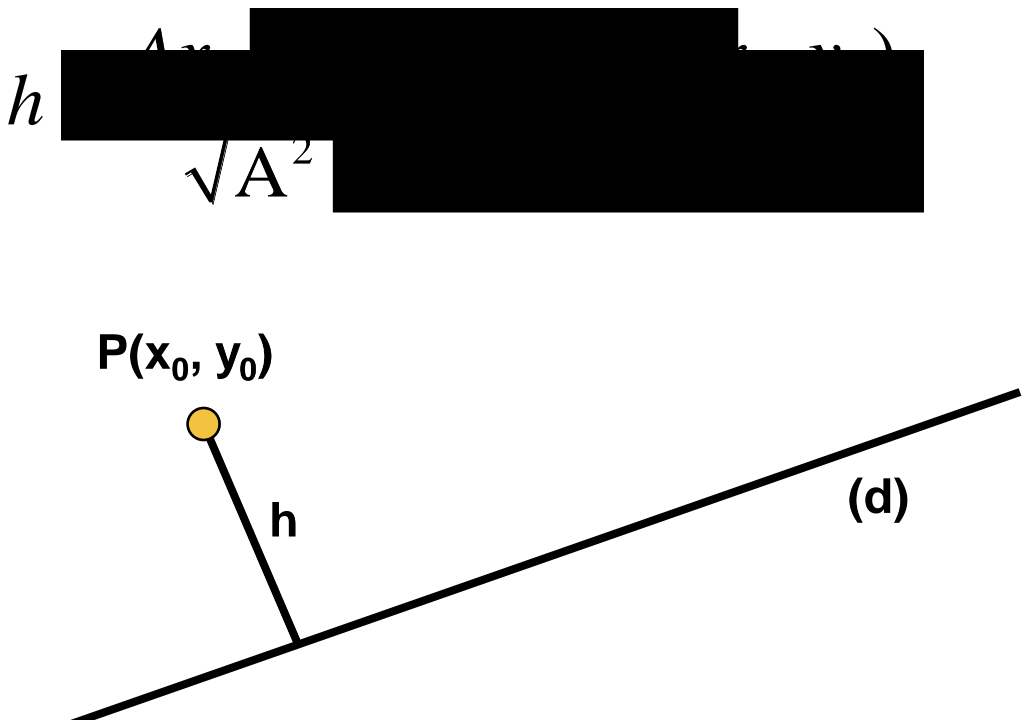
Cấu trúc dữ liệu cơ bản

- Đường thẳng chia mặt phẳng làm 3 phần
  - Phần 1: Gồm các điểm trên đường thẳng F(x,y)=0
  - Phần 2: Gồm các điểm làm cho F(x,y)>0
  - Phần 3: Gồm các điểm làm cho F(x,y)<0



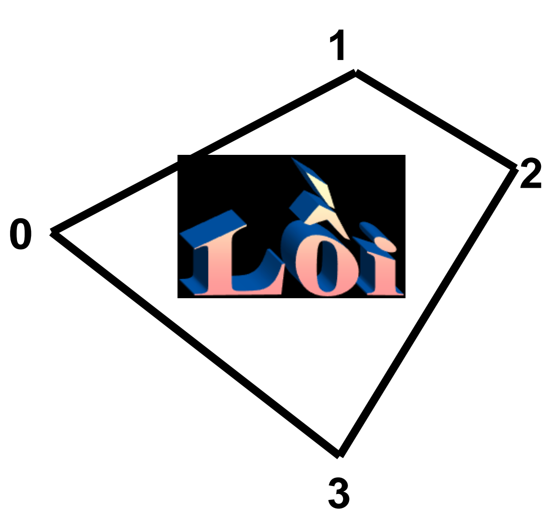
Cấu trúc dữ liệu cơ bản

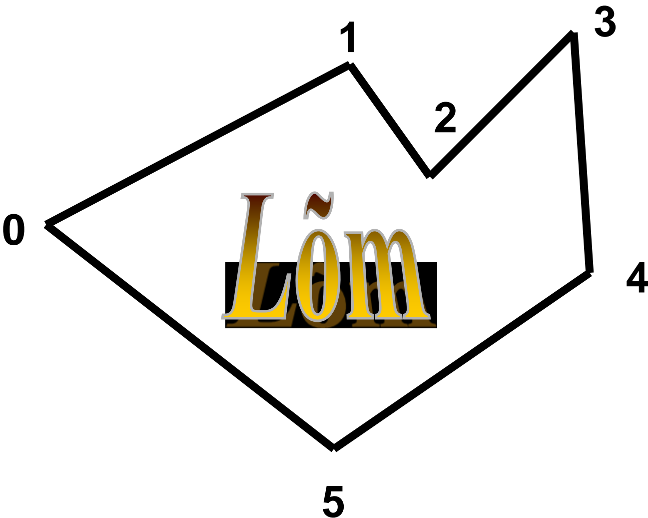
- Khoảng cách từ điểm $P(x_0, y_0)$ đến đường thẳng (d) có phương trình $F(x,y)=Ax+By+C=0$



Cấu trúc dữ liệu cơ bản

- Đa giác: được xác định bởi tập đỉnh được liệt kê thứ tự theo chiều kim đồng (hay ngược chiều kim đồng hồ)
  - Đa giác lồi
  - Đa giác lõm





Cấu trúc dữ liệu cơ bản

```
typedef struct PointTag
{
        double x, y;
} Point;
```

```
typedef struct LineTag
{
        double A, B, C;
} Line;
```

```
#define MAXPOINT 100
typedef struct PolygonTag
{
        Point aPoints[MAXPOINT];
        int n;
} Polygon;
```

Cấu trúc dữ liệu cơ bản

```
void TaoDuongThang(Point p1, Point p2, Line &line)
{
        line.A =
        line.B =
        line.C =
}

double F(Point p, Line line)
{




}
```

Cấu trúc dữ liệu cơ bản

```
double KhoangCachDiemVaDuongThang(Point p, Line line)
{
}
```

Cấu trúc dữ liệu cơ bản

```
bool CungPhia(Point A, Point B, Line line)
{



}
```

Điểm và đoạn thẳng, đường thẳng và tia

- Bài toán 1 [Điểm có thuộc đường thẳng]: Tìm vị trí tương đối giữa điểm $P(x_0, y_0)$ và đường thẳng đi qua 2 điểm $A(x_1, y_1)$ và $B(x_2, y_2)$
- Thuật toán
  - Bước 1: Viết phương trình dưới dạng tổng quát
    - $F(x, y) = Ax+By+C=0$
  - Bước 2: P thuộc đường thẳng AB nếu
    - $F(x_0, y_0) = 0$

Điểm và đoạn thẳng, đường thẳng và tia

```
bool DiemThuocDuongThang(Point p, Point A, Point B)
{












}
```

Điểm và đoạn thẳng, đường thẳng và tia

- Bài toán 2 [Điểm có thuộc đoạn thẳng] : Kiểm tra điểm $P(x_0, y_0)$ có thuộc đoạn thẳng nối 2 điểm $A(x_1, y_1)$ và $B(x_2, y_2)$

- Thuật toán
  - Bước 1: Viết phương trình dưới dạng tổng quát
    - $F(x, y) = Ax+By+C=0$
  - Bước 2: P thuộc đoạn AB nếu thỏa mãn các điều kiện
    - $F(x_0, y_0) = 0$
    - $Min(x_1, x_2) \le x_0 \le Max(x_1, x_2)$
    - $Min(y_1, y_2) \le y_0 \le Max(y_1, y_2)$

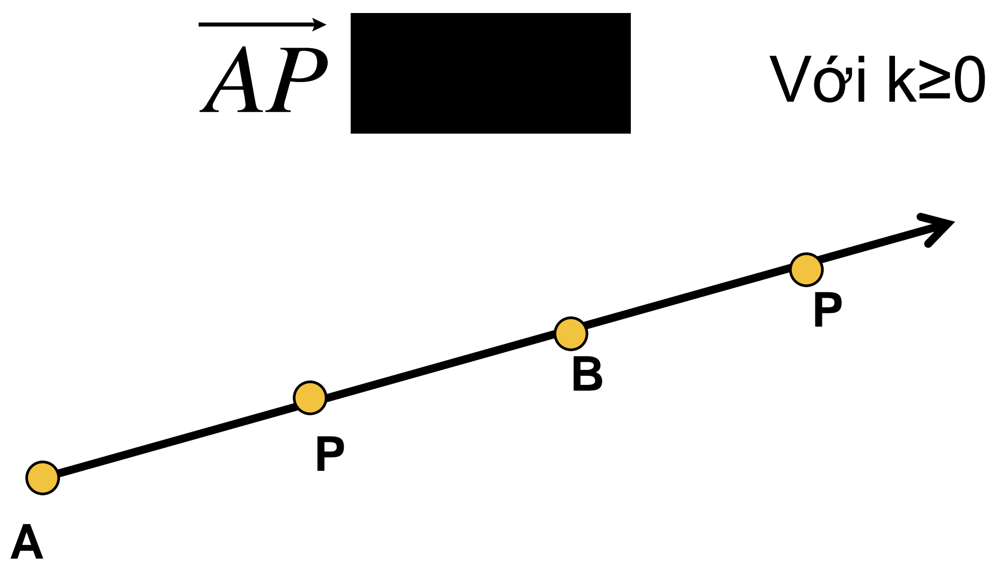
Điểm và đoạn thẳng, đường thẳng và tia

```
bool DiemThuocDoanThang(Point p, Point A, Point B)
{

}
```

Điểm và đoạn thẳng, đường thẳng và tia

- Bài toán 3 [Điểm có thuộc tia] : Kiểm tra điểm $P(x_0, y_0)$ có thuộc tia AB không (trong đó $A(x_1, y_1)$, $B(x_2, y_2)$)

- P thuộc tia AB nếu



Điểm và đoạn thẳng, đường thẳng và tia

- Thuật toán
  - Bước 1: Viết phương trình dưới dạng tổng quát
    - $F(x, y) = Ax+By+C=0$
  - Bước 2: P thuộc tia AB nếu thỏa mãn các điều kiện
    - $F(x_0, y_0) = 0$
    - $(x_0-x_1)(x_2-x_1)\geq 0$
    - $(y_0-y_1)(y_2-y_1)\geq 0$

Điểm và đoạn thẳng, đường thẳng và tia

```
bool DiemThuocTia(Point p, Point A, Point B)
{





}
```

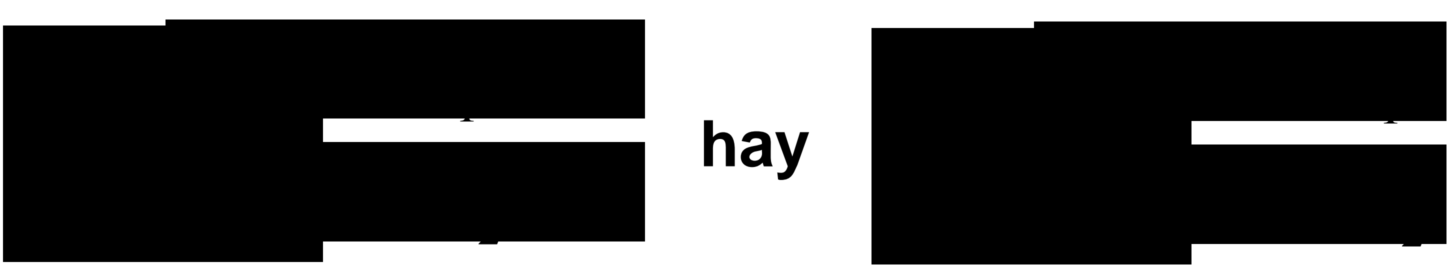
Giao điểm 2 đoạn thẳng, đường thẳng

- Bài toán 4 [Giao điểm 2 đường thẳng]: Tìm giao điểm của 2 đường thẳng có phương trình tổng quát

$A_1 x$

$A_2 x$

- Thuật toán: Giải hệ phương trình



Giao điểm 2 đoạn thẳng, đường thẳng

- Bước 1: Tính d

  dx

  dy

- Bước 2: Biện luận
  - Nếu d=dx=dy=0 thì 2 đường thẳng trùng nhau
  - Nếu d=0 và (dx≠0 hay dy≠0) thì 2 đường thẳng song song
  - Nếu d≠0 thì giao điểm có tọa độ

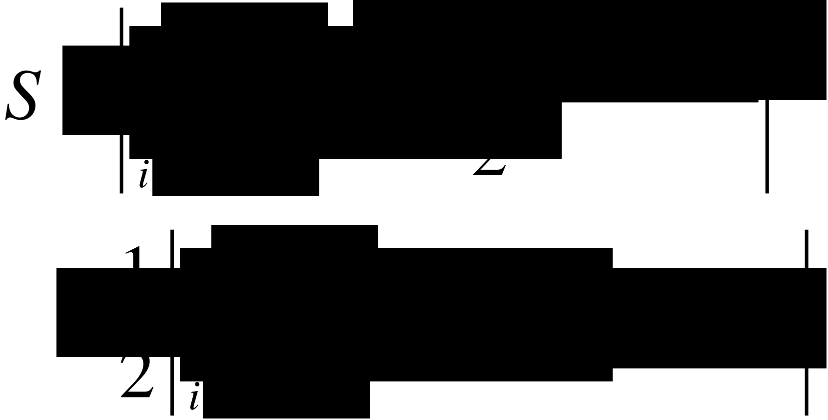
Giao điểm 2 đoạn thẳng, đường thẳng

```
int TimGiaoDiem2DuongThang(Line line1, Line line2,
                           Point &p)
{

}
```

Đa giác

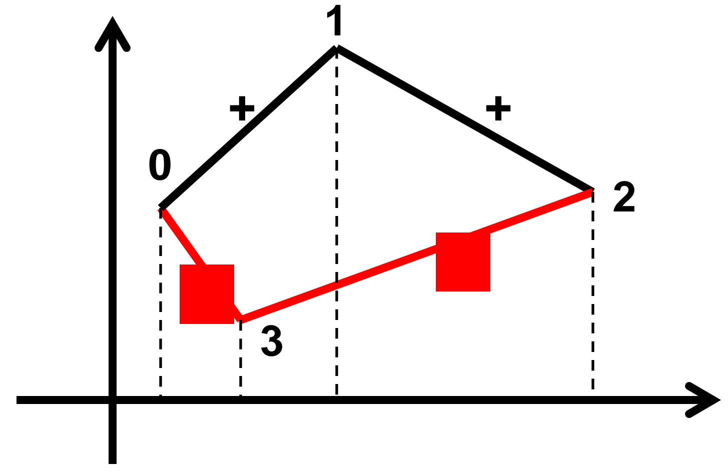
- Bài toán 5 [Diện tích đa giác]: Cho đa giác T. Hãy tính diện tích của đa giác T

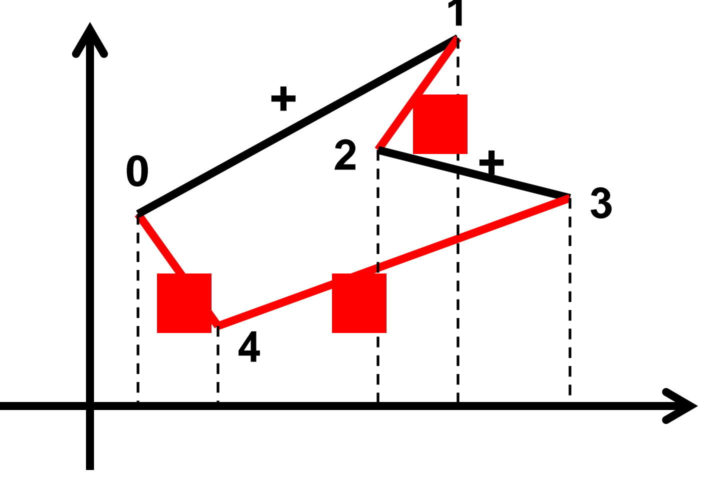
- Thuật toán: Gọi S là diện tích đa giác



  - Chú ý: Đỉnh P_n cũng là đỉnh P_0

Đa giác





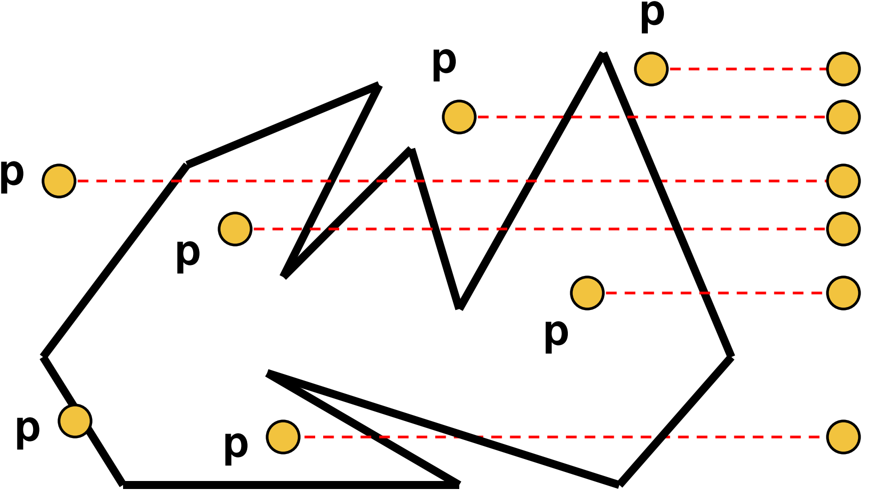
- Ý nghĩa:
  - Đối với đa giác lồi: S bằng HIỆU ■ các hình thang nằm trên với ■ các hình thang nằm dưới
  - Đối với đa giác lõm: S bằng tổng đại số các diện tích của hình thang

Đa giác

```
double TinhDienTichDaGiac(Polygon T)
{









}
```

Đa giác

- Bài toán 6 [Kiểm tra 1 điểm nằm trong hay ngoài đa giác]: Cho đa giác T và điểm P. Hãy kiểm tra xem P thuộc miền trong hay miền ngoài của đa giác T

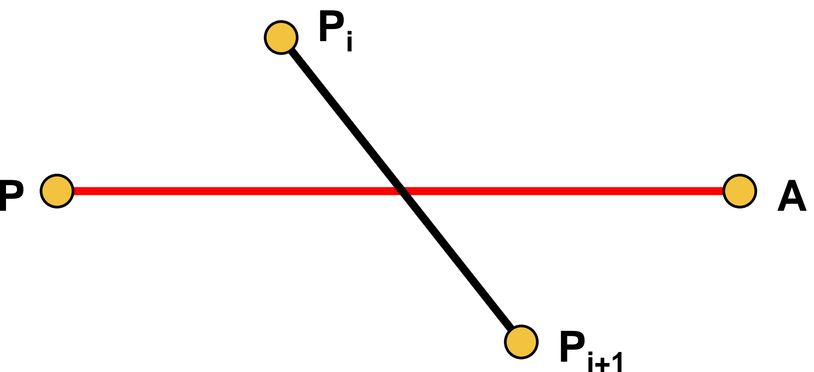


Đa giác

- Thuật toán:
  - Nếu P thuộc bất kỳ cạnh nào của đa giác T thì được xem là thuộc miền trong của đa giác
  - Ngược lại kẻ đoạn thẳng PA song song trục hoành và có hoành độ lớn hơn các hoành độ các điểm (dĩ nhiên lớn hơn hoành độ điểm P)
    - Tính số giao điểm (num) của đoạn thẳng PA với các cạnh đa giác (cũng là các đoạn thẳng)
    - Nếu num lẻ thì P trong đa giác
Ngược lại P nằm ngoài đa giác

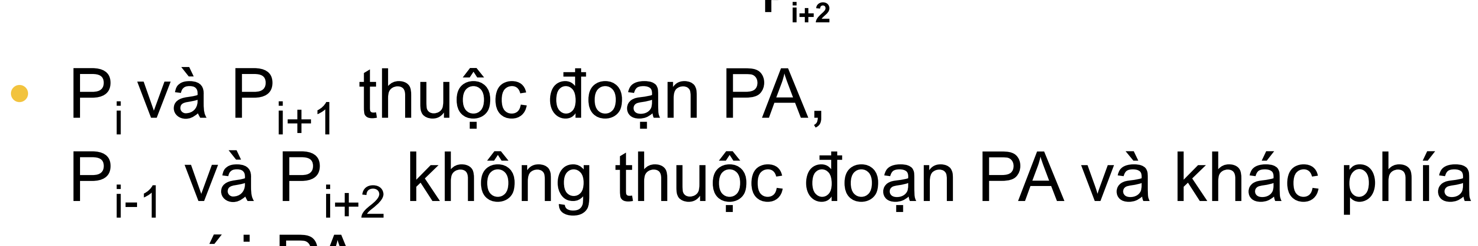
Đa giác

- 3 trường hợp sau được xem như tăng thêm 1 giao điểm
  - Đoạn PA cắt cạnh P_iP_{i+1} và 2 điểm P_i và P_{i+1} không thuộc đoạn thẳng

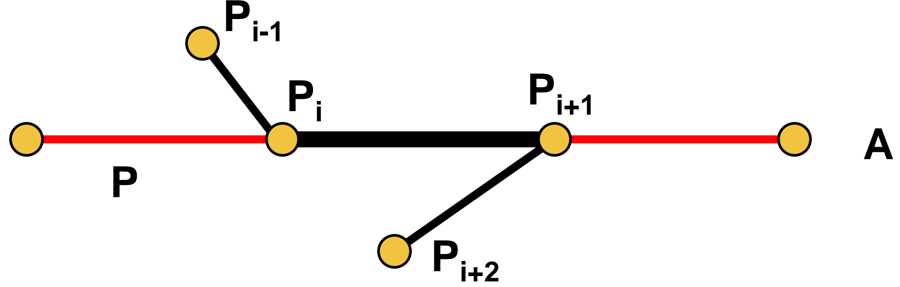


Đa giác

- Điểm P_i không thuộc đoạn PA, P_{i+1} thuộc đoạn PA và 2 điểm P_i và P_{i+2} nằm 2 phía khác nhau so với đoạn PA



- P_i và P_{i+1} thuộc đoạn PA,
  P_{i-1} và P_{i+2} không thuộc đoạn PA và khác phía so với PA

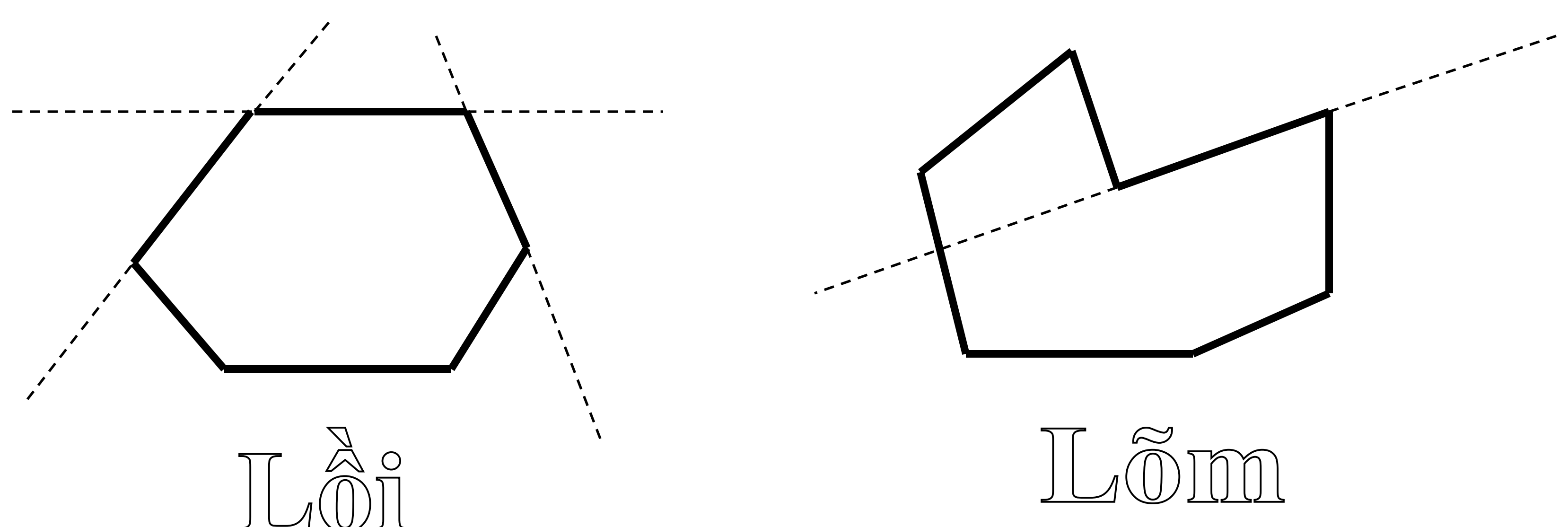


Đa giác

```
int DiemTrongDaGiac(Polygon T, Point P)
{
}
```

Đa giác

- Bài toán 7 [Kiểm tra đa giác lồi]: Cho đa giác T. Hãy kiểm tra xem đa giác T là đa giác lồi hay đa giác lõm



Đa giác

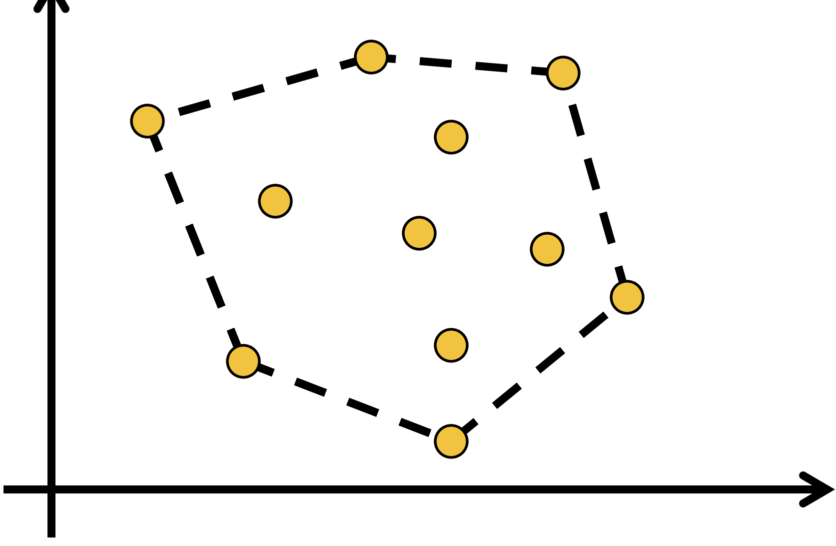
- Thuật toán
  Đa giác T lồi khi
  - Với mỗi cạnh P_iP_{i+1} $(0 \le i < n)$
    – Đỉnh P_{i+2} và đỉnh P_j $(0 \le j < n)$ phải cùng phía so với đường thẳng qua cạnh P_iP_{i+1}
  - Chú ý:
    – Đỉnh P_n được xem như là đỉnh P_0,
    – Đỉnh P_{n+1} được xem như đỉnh P_1

Đa giác

```
int LaDaGiacLoi(Polygon T)
{
















}
```

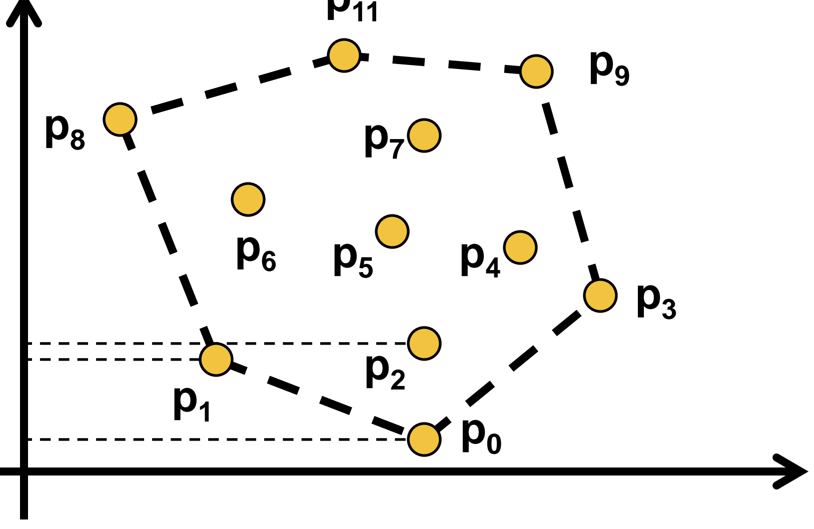
Đa giác

- Bài toán 8 [Bao lồi]: Cho tập điểm P_0, P_1, …, P_{n-1} ($n \le 100$). Hãy tìm đa giác lồi có các đỉnh là một số điểm trong số n điểm đã cho và chứa các điểm còn lại, đồng thời có chu vi nhỏ nhất.



Đa giác

- Thuật toán
  - Bước 1: Sắp xếp các điểm có tung độ tăng dần
  - Bước 2: Chọn đỉnh thứ nhất là đỉnh có tung độ lớn nhất
  - Bước 3 [Lặp]: Giả sử đã chọn được các đỉnh T_0, T_1, …, T_i. Chọn điểm T_{i+1} thỏa điều kiện
    - T_{i+1} chưa được chọn
    - Tập điểm đã chọn nằm về một phía so với đường thẳng qua đoạn T_iT_{i+1}



Đa giác

```
void TimBaoLoi(Point p[], int n, Polygon &T)
{
}
```

Chú ý về lập trình với số thực

- Tránh phép chia: Thay thế phép chia thành phép nhân
- So sánh số thực: Khi so sánh biểu thức E (E chứa số thực) với số 0, chúng ta thường chọn số dương ■ nhỏ cỡ một phần ngàn. Nếu trị tuyệt đối của E nhỏ hơn ■ thì được coi như E bằng 0

```
#define EPS 0.001
…
if (abs(E) < EPS)
{
        …
}
```

Tóm tắt chương 9

Chương 10

NH

Tối ưu hóa chương trình

- 2 Đặc trưng trong chương trình cần tối ưu
  - Tối ưu hóa thời gian thực hiện chương trình
  - Tối ưu hóa không gian lưu trữ dữ liệu

- 2 Loại tối ưu
  - Tối ưu chương trình không làm thay đổi thuật toán (Chỉnh sửa mã chương trình)
  - Tối ưu chương trình làm thay đổi thuật toán

TỐI ƯU HÓA THỜI GIAN CHỈNH SỬA MÃ CHƯƠNG TRÌNH

Chỉnh sửa mã chương trình

- Các cách chỉnh sửa mã chương trình
  - Quy tắc Vòng lặp
  - Quy tắc Logic
  - Quy tắc Hàm
  - Quy tắc Biểu thức

QUY TẮC VÒNG LẶP

Tối ưu câu lệnh lặp

- Quy tắc vòng lặp 1: Đưa code ra ngoài vòng lặp
  - Đưa các tính toán không phụ thuộc vào chỉ số lặp ra khỏi vòng lặp
  - Các biểu thức tính toán nếu đều được tính toán giống nhau qua các lần lặp thì nên được để ngoài vòng lặp
  - Chú ý những biểu thức chứa những phép toán tốn nhiều thời gian: *, /, hàm mũ, lấy căn, ...

Tối ưu câu lệnh lặp

- Ví dụ: Phân tích và Tối ưu đoạn mã sau theo quy tắc trên

```
for (i=0; i<n; i++)
{
        x[i] = x[i] * exp(sqrt(PI/2));
}
```

Tối ưu câu lệnh lặp

- Quy tắc vòng lặp 2: Kết hợp các biểu thức kiểm tra
  - Một vòng lặp hiệu quả sẽ chứa càng ít biểu thức logic dùng để kiểm tra kết thúc vòng lặp càng tốt. Đặc biệt các vòng lặp nằm sâu bên trong
  - Tốt nhất chỉ nên có 1 biểu thức logic kiểm tra kết thúc vòng lặp.
  - Cố gắng thay thế một số điều kiện thoát bằng điều kiện thoát khác hiệu quả hơn

Tối ưu câu lệnh lặp

- Ví dụ 1: Phân tích và Tối ưu đoạn mã sau theo quy tắc trên

```
i=0;
while (i<n && x[i]!=value)
      i++;

if (i<n)
      found = 1;
else
      found = 1;
```

Tối ưu câu lệnh lặp

Tối ưu câu lệnh lặp

- Ví dụ 2: Cho đoạn mã Tìm kiếm phần tử có giá trị value trong mảng đã được sắp xếp tăng. Hãy phân tích và cải tiến để giảm biểu thức logic trong vòng lặp của đoạn mã

```
for (i=0; i<n; i++)
{
        if (x[i]==value)
        {
                found = 1;
                break;
        }
        if (x[i] > value)
        {
                found = 0;
                break;
        }
}
```

Tối ưu câu lệnh lặp

Tối ưu câu lệnh lặp

- Quy tắc vòng lặp 3: Tháo bỏ vòng lặp – Do chi phí thay đổi chỉ số lớn:
  - Trong vòng lặp ngắn hay thân vòng lặp ít code thì chi phí lớn thường nằm trong các lệnh thay đổi chỉ số.
  - Chi phí thay đổi chỉ số vòng lặp thường được giảm bằng cách
    - Bỏ vòng lặp
    - Giảm số lần lặp, tăng số lệnh trong thân vòng lặp

Tối ưu câu lệnh lặp

- Ví dụ 1: Phân tích và cải tiến đoạn mã sau

```
sum=0;
for (i=0; i<10; i++)
        sum = sum + x[i];
```

Tốu ưu câu lệnh lặp

- Ví dụ 2: Hãy cải tiến thuật toán tìm kiếm tuần tự giá trị value trong dãy tăng dần

```
x[n] = value;
i=0;
while (x[i] < value)
      i++;

if (i<n && value==x[i])
      found = 1;
else
      found = 0;
```

Tối ưu câu lệnh lặp

Tối ưu câu lệnh lặp

- Quy tắc vòng lặp 4: Tháo bỏ vòng lặp – Do chi phí phép gán lớn
  - Nếu chi phí của vòng lặp tập trung vào những phép gán thì những phép gán này có thể được bỏ bằng cách:
    - Lặp lại đoạn mã và
    - Thay đổi cách dùng biến

Tối ưu câu lệnh lặp

- Ví dụ: Cho hàm tính số Fibonacci thứ n như sau. Hãy phân tích và cải tiến để hàm sử dụng ít phép gán hơn

```
int Fibonacci(int n)
{
      int f1, f2, f3, i;
      if (n<1 || n>MAXFIBO) return 0;
      if (n<=2) return 1;
      f1 = 1; f2=1;
      for (i=3; i<=n; i++)
      {
            f3 = f1 + f2;
            f1 = f2;
            f2= f3;
      }
      return f3;
}
```

Tối ưu câu lệnh lặp

Tối ưu câu lệnh lặp

- Quy tắc vòng lặp 5: Tổ hợp vòng lặp
  - Nếu 2 vòng lặp gần nhau thực hiện trên cùng tập phần tử thì tổ hợp các lệnh trong thân của 2 vòng lặp và chỉ dùng một vòng lặp

Tối ưu câu lệnh lặp

- Ví dụ: Phân tích và cải tiến thuật toán tìm max và min sau

```
min = a[0];
max = a[0];
for (i=1; i<n; i++)
        if (min > a[i])
                min = a[i];
for (i=1; i<n; i++)
        if (max < a[i])
                max = a[i];
```

Tối ưu câu lệnh lặp

QUY TẮC LOGIC

Quy tắc logic

- Quy tắc logic 1: Tận dụng các biểu thức đại số đồng dạng (tương đương)
- Ví dụ:
  - Thay vì kiểm tra sqrt(X)>0 trong vòng lặp chúng ta sẽ kiểm tra X!=0
  - Thay vì kiểm tra sqrt(X)>sqrt(Y) trong vòng lặp chúng ta sẽ kiểm tra X>Y

Quy tắc logic

- Quy tắc logic 2: Dùng hàm có chu kỳ ngắn
  - Nếu chúng ta muốn kiểm tra một hàm không giảm với một ngưỡng nào đó thì chúng ta không cần phải tính toán hàm tiếp nếu ngưỡng đã đạt đến
- Ví dụ:

```
sum=0;
for (i=0; i<n; i++)
        sum = sum + x[i];
greater = (sum > threshold)
```

Quy tắc logic

```
sum=0;
i=0;

while (i<n && sum<=threshold)
{
        sum = sum + x[i];
        i++;
}

greater = (sum > threshold)
```

Quy tắc logic

```
sum=0;
i=0;
if (n%2==1)
{
        i=1;
        sum=x[0];
}

while (i<n && sum<=threshold)
{
        sum = sum + x[i] + x[i+1];
        i = i+2;;
}
greater = (sum > threshold)
```

Tối ưu biểu thức logic

- Quy tắc logic 3: sắp xếp lại các biểu thức kiểm tra
  - Phép toán logic AND: Biểu thức có khả năng sai nhiều nhất được sắp trước:

$$P_{false}(A_i) \;\blacksquare\; P_{false}(A_{i\blacksquare})$$

```
if (A1 && A2 && … && An)
{
        …
}
```

Tối ưu biểu thức logic

- Quy tắc logic 3: sắp xếp lại các biểu thức kiểm tra
  - Phép toán logic OR: Biểu thức có khả năng đúng nhiều nhất được sắp trước:

$$P_{true}(A_i) \blacksquare P_{true}(A_{i\blacksquare})$$

```
if (A1 || A2 || ... || An)
{
        ...
}
```

TỐI ƯU HÓA THỜI GIAN THAY ĐỔI THUẬT TOÁN

THAY ĐỔI THUẬT TOÁN

- Dùng phương pháp Chia để trị
- Dùng phương pháp Quy hoạch động – Bảng tra (lookup table)
- Tận dụng các công thức

Dùng phương pháp Chia để trị

- Dùng phương pháp Chia để trị
  - Khi chia được bài toán thành các bài toán con giống nhau
  - Chúng ta chỉ cần giải 1 bài toán con
  - Dùng kết quả này cho các bài toán con giống nhau mà không cần giải lại

Dùng phương pháp Chia để trị

- Ví dụ:
  - Tính a^n
  - Tìm max/min của dãy số nguyên
  - Tìm số nhỏ thức k trong mảng nguyên
  - Tìm kiếm nhị phân
  - Quicksort

Dùng phương pháp Quy hoạch động – Bảng tra (lookup table)

- Dùng phương pháp Quy hoạch động – Bảng tra (lookup table)
  - Những giá trị được dùng nhiều lần, chúng ta chỉ cần tính 1 lần rồi lưu lại trong 1 bảng (gọi là bảng lookup)
  - Khi cần giải lại bài toán đã giải chúng ta chỉ tra trong bảng lookup

Dùng phương pháp Quy hoạch động – Bảng tra (lookup table)

- Ví dụ: Tính $C_n^0, C_n^1, C_n^2, \ldots, C_n^n$

$$C_n^k \;\text{[illegible]}\; \frac{n!}{k!(n\;\text{[illegible]}}$$

- Cách 1: Cách thông thường
  - Tính k!
  - Tính tổ hợp: Gọi tính giai thừa
  - Tính các C_n^k

Dùng phương pháp Quy hoạch động – Bảng tra (lookup table)

- Cách 2: Dùng bảng lookup
  - Dùng bảng (n+1) phần tử a[0], a[1], ..., a[n] để tính và lưu a[i]=i!

Tận dụng các công thức

- Sử dụng các công thức thay cho vòng lặp
- Tính toán trên giấy trước khi lập trình
- Tìm mối quan hệ giữa bước tính trước và bước tính sau

Tận dụng các công thức

- Ví dụ: Tính tổng S sau:

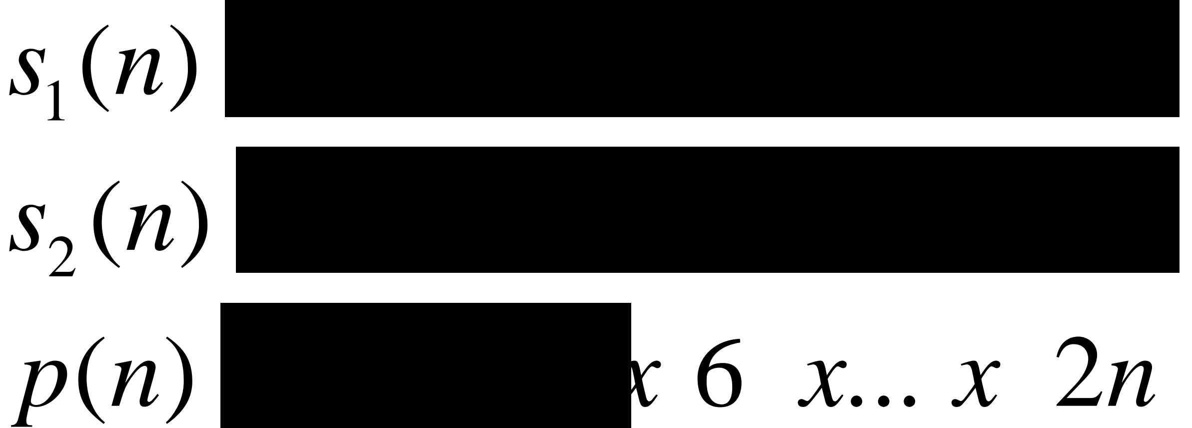
$$s(n)$$

```
int TinhTong(int n)
{
        int s;
        s=0;

        int i;
        for (i=1; i<=n; i++)
              s = s + i;

        return s;
}
```

```
int TinhTong(int n)
{
        int s;

        s = n*(n+1)/2;

        return s;
}
```

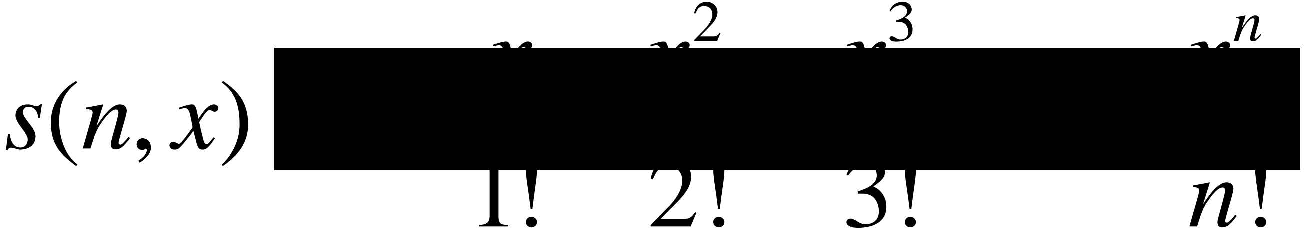
Tận dụng các công thức

- Ví dụ: Viết chương trình tính các biểu thức sau:



Tận dụng các công thức

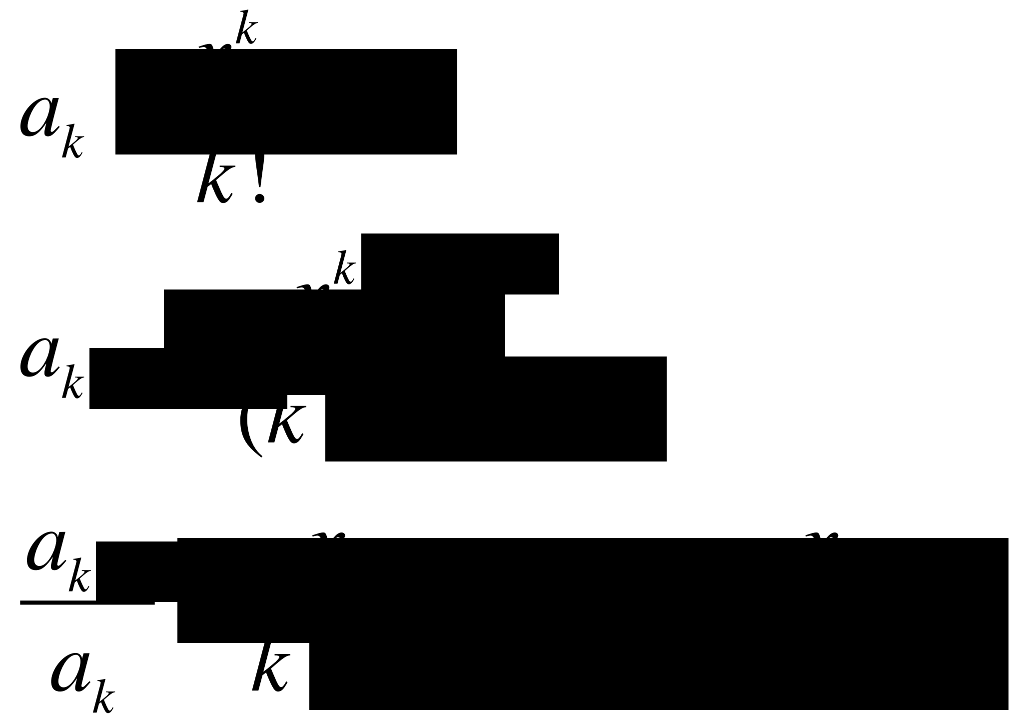
- Ví dụ: Viết chương trình tính tổng



- Cách 1: Cách thông thường
  - Chia nhỏ vấn đề, cài đặt các hàm cho từng vấn đề
    - Tính x^k
    - Tính k!

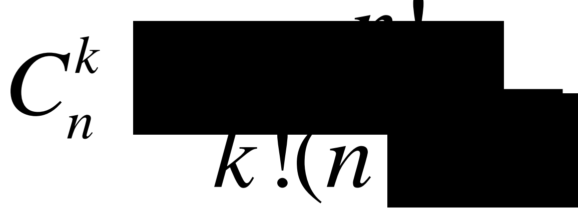
Tận dụng các công thức

- Cách 2: Tìm mối quan hệ giữa bước tính trước và bước tính sau



Tận dụng các công thức

- Ví dụ: Tính $C_n^0, C_n^1, C_n^2, \ldots, C_n^n$



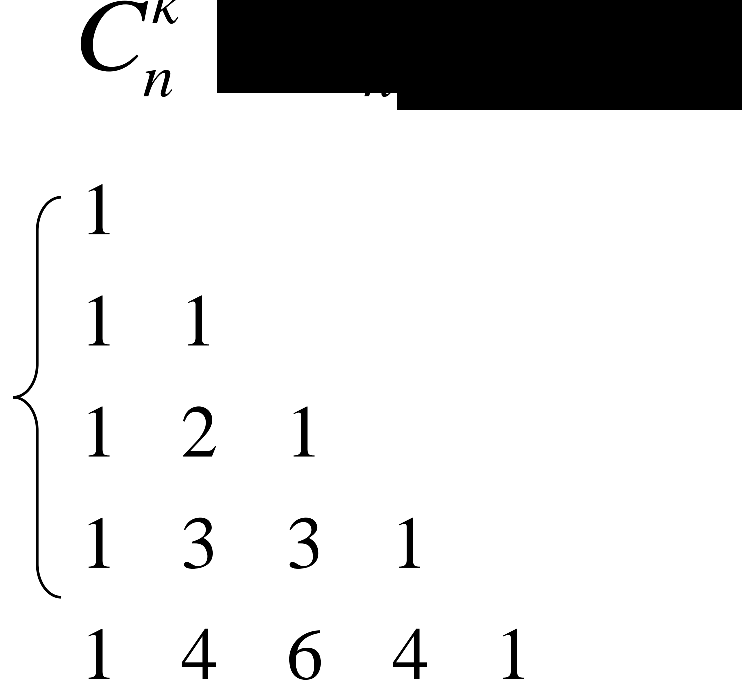
- Cách 1: Cách thông thường
  - Tính k!
  - Tính tổ hợp: Gọi tính giai thừa
  - Tính các C_n^k

Tận dụng các công thức

- Cách 2: Dùng bảng lookup
  - Dùng bảng (n+1) phần tử a[0], a[1], ..., a[n] để tính và lưu a[i]=i!

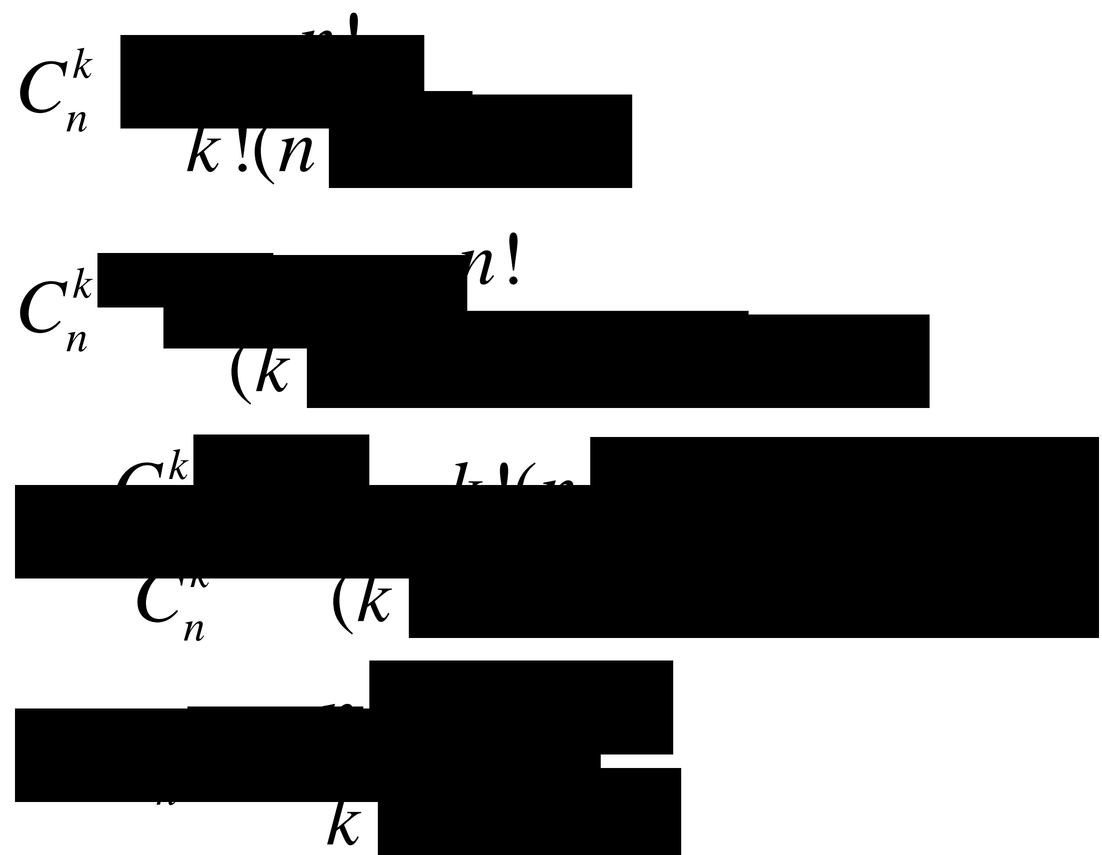
Tận dụng các công thức

- Cách 3: Dùng quy nạp



Tận dụng các công thức

- Cách 4: Tìm mối quan hệ giữa bước tính trước và bước tính sau



Tận dụng các công thức

- Cách 5:
  - Hạ xuống phân nửa cho mỗi cách (2), (3), (4) do tính đối xứng

$$C_n^k$$

  - Chỉ cần tính với k=1, 2,..., [n/2]

Tận dụng các công thức

- Ví dụ: Kiểm tra n có là số nguyên tố không
- Cách 1: Dựa vào định nghĩa số nguyên tố
  - Kiểm n có chia hết cho các số từ 2→n-1 không
- Cách 2: Nhận xét các ước số của n chỉ nằm trong [2...n/2]
  - Kiểm tra n có chia hết cho các số từ 2→n/2 không

Tận dụng các công thức

- Cách 3: Nhận xét nếu n không nguyên tố thì sẽ tồn tại ước số nguyên tố thuộc [2...sqrt(n)]
  - Nếu n<2 thì n không là số nguyên tố
  - Nếu n=2 thì n là số nguyên tố
  - Kiểm tra n có là số chẵn không
  - Kiểm tra n có chia hết cho các số từ 3→sqrt(n) không: 3,5,7, ..., sqrt(n)