ĐẠI HỌC QUỐC GIA TP.HCM

TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN

ĐỒ ÁN CUỐI KÌ

PHÂN TÍCH VÀ THIẾT KẾ THUẬT TOÁN

KHOA: KHOA HỌC MÁY TÍNH

ĐỀ TÀI: DYNAMIC PROGARMMING

GV hướng dẫn: ThS. Huỳnh Thị Thanh Thương

Nhóm thực hiện:

1. Nguyễn Đức Anh Phúc – 20520276
2. Ngô Văn Tấn Lưu – 20521591
3. Huỳnh Viết Tuấn Kiệt – 20521494
4. Trương Thành Thắng – 20521907

MỤC LỤC

GIỚI THIỆU VỀ QUY HOẠCH ĐỘNG VÀ CÁC BÀI TOÁN ỨNG DỤNG

01.01 Giới thiệu về quy hoạch động

01.01.01 Phương pháp quy hoạch động

Quy hoạch động vừa là một phương pháp tối ưu hóa toán học vừa là một kỹ thuật lập trình máy tính. Phương pháp này được phát triển bởi Richard Bellman vào những năm 1950 và đã được ứng dụng trong nhiều lĩnh vực, từ kỹ thuật hàng không vũ trụ đến kinh tế. Trong phạm vi đồ án này, nhóm em chỉ nói về Quy hoạch động là một kỹ thuật lập trình.

01.01.02 Dấu hiệu sử dụng Quy hoạch động

Có 2 dấu hiệu chính mà một bài toán phải có để áp dụng kỹ thuật quy hoạch động là: optimal substructure và overlaping subproblems.

- Overlapping subproblems: Việc giải bài toán con được lặp lại nhiều lần.
- Optimal substructure: Nếu một vấn đề có thể được giải quyết một cách tối ưu bằng cách chia nó thành các bài toán con và sau đó tìm ra lời giải tối ưu cho các bài toán con, thì nó được cho là có cấu trúc con tối ưu.

01.01.03 Ý tưởng quy hoạch động

- Tạo ra một bảng phương án để lưu trữ kết quả của các bài toán đã được giải.
- Sử dụng kết quả đã lưu trong bảng mà không cần giải lại.

01.01.04 Các bước thực hiện

Bước 1: Phân tích đặc trưng optimal substructure, overlapping subproblems.
Bước 2: Xác định phương trình quy hoạch động.
Bước 3: Tạo bảng và lưu giá trị.
Bước 4: Tra bảng, xây dựng lời giải ban đầu.

01.01.05 Nhận xét

01.01.05.1 Ưu điểm

- Mỗi bài toán chỉ giải 1 lần → Thực hiện nhanh.
- Thường được vận dụng để giải các bài toán tối ưu, bài toán có công thức truy hồi.

01.01.05.2 Nhược điểm

- Không phải cứ sự kết hợp của bài toán con là cho lời giải bài toán ban đầu.
- Không hiệu quả khi bài toán không có công thức truy hồi.
- Không hiệu quả khi số lượng bài toán con rất lớn.

01.02 Các bài toán ứng dụng

Trong nội dung đồ án này, nhóm em trình bày lời giải các bài toán ứng dụng sau:

1. Fibonacci
2. Prefix Caculation
3. Dynamic Programming on Grid
4. Longest Increasing Subsequences
5. Knapsack
6. Longest Common Subsequence
7. Dynamic Programming on Tree
8. Sparse Table
9. Matrix Exonentiation
10. Dynamic Programming with Bitmask
11. Knuth's Optimization

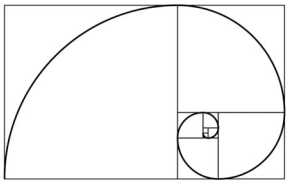
01.03 Bài toán 1: Fibonacci

01.03.01 Mô tả bài toán

Cho số nguyên dương n. Tìm số Fibonacci thứ n.

10 số fibnacci đầu tiên là :

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55$$



01.03.02 Mô hình hóa bài toán

01.03.02.1 Input:

- 1 dòng duy nhất chứa một số nguyên dương n là số Fibonacci thứ n

01.03.02.2 Output:

- 1 dòng duy nhất chứ số Fibonacci thứ n
- Lưu ý: số thứ tự của số Fibonacci bắt đầu từ 1.

01.03.02.3 Ví dụ:

Input	Output
7	13

01.03.03 Thiết kế thuật toán

- Công thức truy hồi số Fibonacci:
- Gọi $fibo(n)$ là số Fibonacci thứ n:

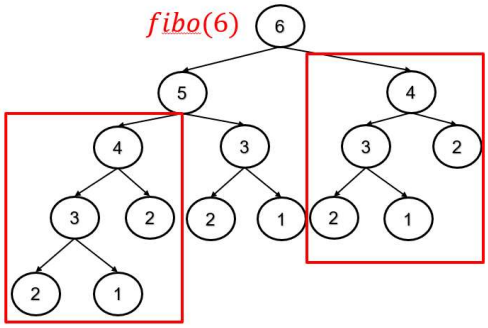
$$fibo(n) = fibo(n-1) + fibo(n-2)$$

- Dựa vào công thức truy hồi, ta có mã giả sau:

```
fibo (n):
        if n == 1 or n == 2:
                        return 1
        return fibo(n - 1) + fibo(n - 2)
```

- Độ phức tạp: $O(2^n)$

– Lý do: Overlapping Subproblem (giải nhiều lần một bài toán con) → Nhược điểm của chia để trị



– Giải quyết bài toán lớn bằng cách kết hợp lời giải của các bài toán con (giống chia để trị, khắc phục nhược điểm chia để trị).
  + Tạo ra một bảng phương án để lưu trữ kết quả của các bài toán đã được giải.
  + Sử dụng kết quả đã lưu trong bảng mà không cần giải lại.

– Phương án mã giả:

```
fib[]
fibo (n):
        if fib[i] != undefined:
                return fib[n]
        if n==1 or n==2:
                fib[n] = 1
        else
                tmp1 = fibo(n-1)
                tmp2 = fibo(n-2)
                fib[n] = tmp1 + tmp2
        return fib[n]
```

– Ngoài ra, còn có cách tiếp cận bằng phương pháp bottom-up

```
fig[]
fib[1] = fib[2] = 1
        for (i = 3; i <= n - 1; i += 1):
                fib[i] = fib[i - 1] + fib[i - 2]
fibo(n):
        return fib[n]
```

01.03.04 Ví dụ minh họa

Input	Output
14	377

```
14
377
D:\SourceCode\SourceMVS\CompetitiveProgramming\Debug\fibonacci.exe (process 16828) exited with code 0.
Press any key to close this window . . .
```

Input	Output
13	233

```
13
233
D:\SourceCode\SourceMVS\CompetitiveProgramming\Debug\fibonacci.exe (process 16224) exited with code 0.
Press any key to close this window . . .
```

```
34
5702887
D:\SourceCode\SourceMVS\CompetitiveProgramming\Debug\fibonacci.exe (process 6652) exited with code 0.
Press any key to close this window . . .
```

Input	Output
34	5702887

01.03.05 Phân tích độ phức tạp

– Cả 2 cách tiếp cận top-down hay bottom-up thì độ phức tạp để xây dựng bảng phương án cho bài toán Fibonacci là $O(n)$ cho thời gian và không gian bộ nhớ là $O(n)$

01.04 Bài toán 2: Prefix Calculation

01.04.01 Đặt vấn đề

01.04.01.1 Bài toán 1 truy vấn

- Mô tả bài toán:
  + Cho dãy a gồm n phần tử được đánh thứ tự từ $1 \rightarrow n$. Cho 1 câu truy vấn $[l, r]$, tìm tổng các phần tử dãy a trong đoạn $[l, r]$.
- Mô hình hóa bài toán:
  + Input:
    - Cho số nguyên n
    - Cho dãy a chứa các phần tử a_i với $i = 1,2,3 \dots n$
    - Cho hai số nguyên: l và r $(1 \leq l \leq r \leq n)$
  + Output:
    - Kết quả $S = a[l] + a[l+1] + a[l+2] + \cdots + a[r-1] + a[r]$
- Ví dụ:

Input	Output
12 2 3 5 2 6 9 4 3 6 9 1 1 3 9	35

- Giải pháp đơn giản
  + Sử dụng vòng lặp, lặp qua dãy a từ vị trí l đến vị trí r, cộng dồn giá trị tại mỗi vị trí và lưu vào biến kết quả.
- Độ phức tạp: $O(n)$

01.04.01.2 Đặt vấn đề

- Cũng với mô tả bài toán như trên, nhưng thay 1 truy vấn thành q truy vấn. Nếu sử dụng vòng lặp để giải quyết, với mỗi câu truy vấn ta phải duyệt lại dãy để tính tổng. Độ phức tạp là $O(n \times q)$ và thuật toán này không hiệu quả trong trường hợp n và q đồng thời cực lớn (với n là kích thước dãy và q là số lượng truy vấn). Do đó, cần có một cách tiếp cận khác khả thi hơn.

01.04.02 Mô tả bài toán

- Cho dãy a gồm n phần tử được đánh thứ tự từ $1 \rightarrow n$. Cho q câu truy vấn $[l, r]$, với mỗi truy vấn, tìm tổng các phần tử dãy a trong đoạn $[l, r]$.

01.04.03 Mô hình hóa bài toán

01.04.03.1 Input:

- Số nguyên dương n: Số lượng phần tử của dãy a
- Số nguyên dương q: Số lượng câu truy vấn
- n phần tử dãy a, $a_i \in R$ là giá trị tại vị trí i trong dãy a $(i = 1,2,3, \dots n)$
- q dòng, mỗi dòng hai số nguyên l và r $(1 \leq l \leq r \leq n)$, vị trí thứ l và r trong dãy a

01.04.03.2 Output

- q dòng, mỗi dòng tương ứng với kết quả 1 truy vấn

$$a[l,r] = a[l] + a[l+1] + \cdots + a[r-1] + a[r]$$

01.04.04 Thiết kế thuật toán

01.04.04.1 Ý tưởng

- Từ dãy a đầu vào có n phần tử
- Sử dụng mảng cộng dồn $prefix\_sum$ (p) với $p[i] = a[1] + a[2] + \cdots + a[i]$
- Tổng $a[l,r] = a[l] + a[l+1] + \cdots + a[r] = p[r] - p[l-1]$

01.04.04.2 Phân tích đặc trưng Optimal Substructure, Overlapping Subproblem

- Gọi $a[l,r]$ là tổng các phần tử dãy a từ vị trí l đến r

$$a[l,r] = a[l] + a[l+1] + \cdots + a[r-1] + a[r]$$

  + Cho 2 truy vấn:

    (1) $a[i,j] = a[i] + a[i+1] + \cdots + a[j]$

    (2) $a[i-1,j] = a[i-1] + a[i] + a[i+1] + \cdots + a[j]$

  + Việc sử dụng vòng lặp, lặp qua phép tính $a[i] + a[i+1] + \cdots + a[j]$ trong truy vấn (2) là không cần thiết, vì phép tính đã được thực thi ở truy vấn (1). Truy vấn (2) có thể được biểu diễn lại như sau:

    (2) $a[i-1,j] = a[i-1] + a[i,j]$

    ⇒ Đây là đặc trưng Overlapping Subproblems

- Dãy $prefix\ sum$ có $p[i] = p[i-1] + a[i]$

  ⇒ Đây là đặc trưng Optimal Substructure

01.04.04.3 Xác định phương trình quy hoạch động

- Gọi $p[i]$ là tổng các phần tử của dãy a từ vị trí 1 đến i $(1 \leq i \leq n)$

$$p[i] = a[1] + a[2] + \cdots + a[i-1] + a[i]$$

  Và

$$p[i] = p[i-1] + a[i] \ (3)$$

– Bài toán cơ sở

$$p[0] = 0 \ (4)$$

theo định nghĩa, $p[i]$ là tổng các phần tử của dãy a từ 1 đến i, nên $p[1] = a[1]$, mà từ $(1) \Rightarrow p[1] = p[0] + a[1]$, như vậy thỏa điều kiện (2)

– Từ (3) và (4), xây dựng được phương trình quy hoạch động

$$p[i] = \begin{cases} 0 & khi\ i = 0 \\ p[i-1] + a[i] & khi\ i \neq 0 \end{cases}$$

01.04.04.4 Tạo bảng và lưu giá trị

– Từ dãy a đầu vào có n phần tử

– Tạo bảng lưu trữ, gọi là mảng p (viết tắt của $prefix$) kích thước bằng mảng a, giá trị $p[i]$ bằng tổng các giá trị phần tử của dãy a từ vị trí i đến vị trí n

```
function prefix_sum(a, n)
     init array p has size = n+1
     p[0] = 0
     for i = 1 -> n do
          p[i] = p[i - 1] + a[i]
     return p
```

01.04.04.5 Tra bảng và xây dựng lời giải

– Từ mỗi truy vấn q với hai giá trị l, r, ta có kết quả mỗi truy vấn

$$q[l, r] = p[r] - p[l-1]$$

trong đó p là dãy $prefix$ đã khởi tạo ở mục **02.01.04.3**

```
function query(p, l, r)
     return p[r] - p[l - 1]
```

01.04.05 Ví dụ minh họa

Input
12 5 2 3 5 2 6 9 4 3 6 9 1 1 3 9 2 6 2 4 1 4 1 1

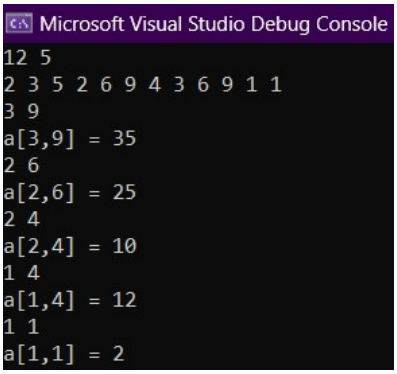
Testcase 1

Input
8 4 3 2 4 5 1 1 5 3 2 4 5 6 1 8 3 3

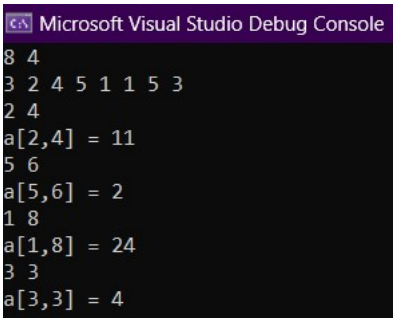
Testcase 2

Input
6 4 1 1 1 1 1 1 1 1 1 3 1 5 2 6

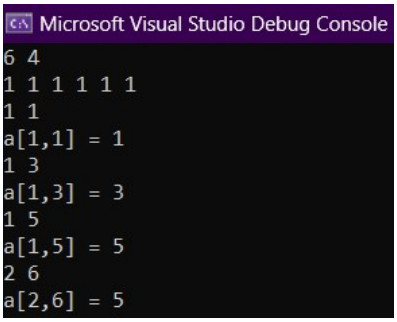
Testcase 3



Kết quả testcase 1



Kết quả testcase 2



Kết quả testcase 3

01.04.06 Phân tích độ phức tạp

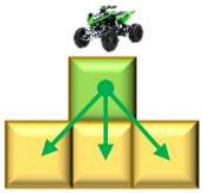
- Tạo mảng $prefix$ cần duyệt qua n phần tử mảng a: $O(n)$
- Tiếp theo duyệt qua q truy vấn: $O(q)$
- Mỗi truy vấn chỉ thực hiện tính toán thông qua truy xuất giá trị từ mảng $prefix$: $O(1)$
- Độ phức tạp thuật toán: $O(n+q)$

01.05 Bài toán 3: Dynamic Programming on Grid

01.05.01 Mô tả bài toán

Chặng cuối cùng trước khi cán đích của một cuộc đua xe địa hình là một vùng chướng ngại hình chữ nhật kích thước $n \times m$ ô (n hàng và m cột). Các tay đua có thể và chỉ có thể vào ô bất kỳ ở hàng đầu tiên và ra khỏi vùng chướng ngại từ ô bất kỳ ở hàng cuối cùng. Xe không được ra khỏi vùng chướng ngại trước khi tới được hàng cuối cùng. Từ một ô (i, j) xe có thể di chuyển sang một trong số các ô $(i + 1, j - 1), (i + 1, j), (i + 1, j + 1)$ nếu ô đó tồn tại. Thời gian vượt qua ô (i, j) là a_{ij}. ($i \in [1, n]$, $j \in [1, m]$).

Hãy xác định thời gian nhỏ nhất một tay đua có thể vượt qua vùng chướng ngại.



6	5	1	3
2	8	7	8
9	4	6	4

6	5	1	3
2	8	7	8
9	4	6	4

11

01.05.02 Mô hình hóa bài toán

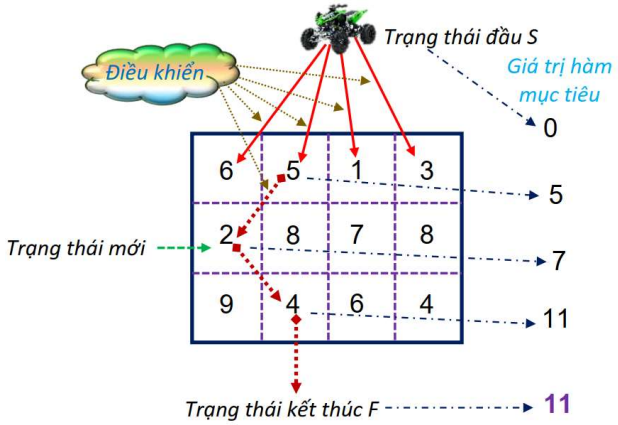
- Input:
  - Dòng đầu tiên chứa 2 số nguyên n và m
  - Dòng thứ i trong n dòng sau chứa m số nguyên $a_{i1}, a_{i2}, \ldots, a_{im}$
- Output
  - Dòng duy nhất chứa độ dài đường đi ngắn nhất

Input	Output
3 4 6 5 1 3 2 8 7 8 9 4 6 4	11

01.05.03 Thiết kế thuật toán

- Phân tích
  - Trạng thái: Vị trí của xe được gọi là trạng thái

+ Hàm mục tiêu: Chi phí thời gian để xe tới và vượt qua được một trạng thái nào đó (tức là tới một nào đó và đi qua khỏi ô đó) được gọi là giá trị hàm mục tiêu
+ Điều khiển: Từ một trạng thái xe phải di chuyển tới trạng thái mới, việc lựa chọn một cách di chuyển được gọi là điều khiển
+ Trạng thái đầu S: Xe ở ngoài vùng chướng ngại chuẩn bị vào
+ Trạng thái kết thúc F: Xe đã qua vùng chướng ngại và đang ở vùng này



– Nguyên lí tối ưu Bellman

> *Không phụ thuộc vào trạng thái ban đầu và điều khiển ban đầu, điều khiển ở bước tiếp theo phải là tối ưu đối với trạng thái nhận được bởi điều khiển ở bước trước.*

– Điều khiển ở bước tiếp theo là tối ưu của điều khiển bước phía trước
  + $p_2 = min(q_1, q_2, q_3) + cost\ (p_2)$
– **Optimal substructure**
  + Để giải bài toán tối ưu, ta chia bài toán đó ra thành các bài toán con tối ưu có kích thước nhỏ hơn và giải chúng
– Gọi $d_{i,j}$ là độ dài đường đi ngắn nhất với vị trí bắt đầu bất kì ở dòng đầu tiên và kết thúc tại ô (i,j)
– Công thức truy hồi:

$$d_{i,j} = min\left(d_{i-1,j}, d_{i-1,j-1}, d_{i-1,j+1}\right) + cost(i,j)$$

– Bài toán cơ sở:

$$d_{1,:} = cost_{1,:}$$

Cost

6	5	1	3
2	8	7	8
9	4	6	4

d

6	5	1	3
7	9	8	9
16	11	14	12

– Mã giả:

```
d[n][m]={INT_MAX}
gridDP(n, m):
        for (j = 1; j <= m; j += 1):
                d[1][j] = cost[1][j]

        for (i = 2; i <= n; i+= 1)
                for (j = 1; j <= m; j += 1):
                        d[i][j] = d[i - 1][j]
                        if j - 1 >= 1:
                                d[i][j] = min(d[i][j], d[i - 1][j - 1])
                        if j + 1 <= m:
                                d[i][j] = min(d[i][j], d[i - 1][j + 1])
                        d[i][j] += cost[i][j]

        result = INT_MAX
        for (j = 1; j <= m; j++)
                result = min(result, d[n][j])
        return result
```

01.05.04 Ví dụ minh họa

Input	Output
3 4 6 5 1 3 2 8 7 8 9 4 6 4	11

```
3 4
6 5 1 3
2 8 7 8
9 4 6 4
11
D:\SourceCode\SourceMVS\CompetitiveProgramming\Debug\dpgrid.exe (process 22388) exited with code 0.
Press any key to close this window . . .
```

Input	Output
4 4 7 9 1 2 6 5 3 1 3 5 9 8 9 4 6 4	13

```
4 4
7 9 1 2
6 5 3 1
3 5 9 8
9 4 6 4
13
D:\SourceCode\SourceMVS\CompetitiveProgramming\Debug\dpgrid.exe (process 2608) exited with code 0.
Press any key to close this window . . .
```

01.05.05 Phân tích độ phức tạp

- Để xây dựng bảng phương án và trích xuất kết quả bằng kĩ thuật quy hoạch động:
  + Độ phức tạp thuật toán $O(N \times M)$
  + Độ phức tạp không gian $O(N \times M)$

01.06 Bài toán 4: Longest Increasing Subsequences

01.06.01 Mô tả bài toán

Cho dãy các số nguyên $A = a_1, a_2, a_3, \dots, a_n$ gồm n phần tử, tìm độ dài của dãy con có độ dài lớn nhất của A được sắp xếp theo thứ tự tăng dần. Cho biết một dãy con của A là một cách chọn ra trong A một số phần tử và giữ nguyên thứ tự.

01.06.02 Mô hình hóa bài toán

01.06.02.1 Input:

- Dòng thứ nhất chứa một số nguyên dương n là độ dài của dãy A đầu vào.
- Dòng thứ hai chứa n số, là các phần tử trong dãy A.

01.06.02.2 Output:

- Dòng thứ nhất chứa một số nguyên dương là độ dài lớn nhất của dãy con tăng
- Dòng thứ hai chứa các phần tử của dãy con tăng đó.
- Lưu ý: số thứ tự trong mảng bắt đầu từ 1.

01.06.02.3 Ví dụ:

Input	Output
12 1 3 18 9 6 2 7 15 2 10 13 3	6

- Giải thích: dãy con tăng dài nhất trong mảng đã cho là: 1 3 6 7 10 13. Dãy này có 6 phần tử.

01.06.03 Thiết kế thuật toán

- Gọi $d[i]$ là độ dài của dãy con tăng dài nhất kết thúc tại i ($\forall i: 1 \leq i \leq n$). Vì dãy con tăng dài nhất chưa biết kết thúc ở đâu trên đoạn từ 1 đến n nên ta sẽ tính toàn bộ $d[i]$ ($\forall i: 1 \leq i \leq n$) để tìm ra ở mỗi vị trí trên mảng, dãy con tăng dài nhất đến vị trí đó là bao nhiêu từ đó tìm ra độ dài của dãy con tăng dài nhất trên toàn bộ mảng.
- Cách làm này có hai tính chất:
  - Ở mỗi index i, cần tìm dãy con tăng dài nhất kết thúc tại i, đây là tính chất **Overlapping Subproblem.**
  - Để tìm được dãy con tăng dài nhất kết thúc tại i cần tìm được dãy con tăng dài nhất kết thúc ở đâu đó trước i. Đây là tính chất **Optimal Substructure.**
- Bài toán cơ sở: $d[1] = 1$. Vì tại vị trí có $index = 1$, dãy con này chỉ có một phần tử.
- Dãy con tăng dài nhất kết thúc ở vị trí i sẽ được thành lập bằng cách lấy a_i ghép vào đuôi của một trong số những dãy con tăng dài nhất bắt đầu tại vị trí a_j đứng trước a_i. Vì $a[i]$ chỉ được ghép vào đuôi những dãy con bắt đầu tại $a[j]$ nào đó nhỏ hơn $a[i]$ để đảm bảo được tính tăng và ta sẽ phải chọn dãy dài nhất để ghép $a[i]$ vào đầu. Như vậy, $d[i]$ có thể được tính như sau:
  - Xét tất cả các chỉ số j trong khoảng từ 1 đến i mà $a[j] < a[i]$, chọn ra chỉ số j có $d[j]$ lớn nhất gọi là $jmax$, sau đó tính $d[i] = d[jmax] + 1$. Đây cũng là công thức truy hồi.
- Từ ý tưởng trên, ta có mã giải như sau:

```
def LIS()
        Initialize array d which has n + 1 elements;
        d[1] = 1;
        for (i: 2 -> n)
        {
                jmax = 0;
                for (j: 1 -> i - 1)
                        if (A[j]<A[i] && d[j]>d[lc])
                                jmax = j;
                d[i] = d[jmax] + 1;
        }
        return maxElement(d);
```

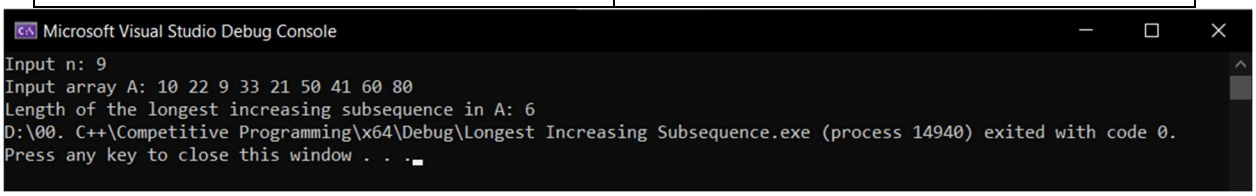
01.06.04 Ví dụ minh họa

- Test case 1:

Input	Output
12 1 3 18 9 6 2 7 15 2 10 13 3	6

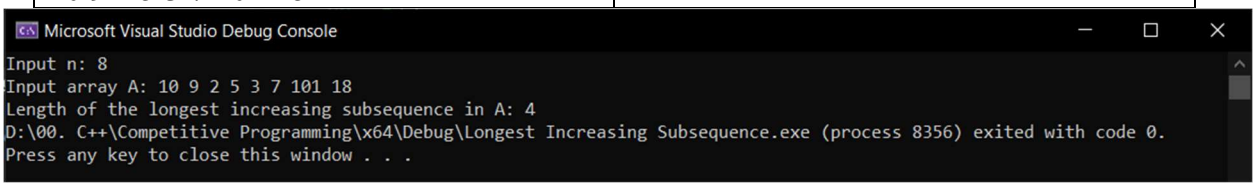
– Test case 2:

Input	Output
9 10 22 9 33 21 50 41 60 80	6



– Test case 3:

Input	Output
8 10 9 2 5 3 7 101 18	4



01.06.05 Phân tích độ phức tạp

Để xây dựng được dãy d, ta cần vòng lặp thứ nhất lặp qua từng phần tử của dãy d. Ở mỗi phần tử của dãy d, để tìm được $jmax$ ta cần lặp qua từ 1 đến $i-1$. Mà độ dài của mảng d là n. Do vậy, độ phức tạp của ý tưởng này là $O(n*n)$.

01.07 Bài toán 5: Knapsack

01.07.01 Mô tả bài toán

Cho n đồ vật và một cái ba lô có thể đựng trọng lượng tối đa M, mỗi đồ vật i có trọng lượng w_i và giá trị là p_i.

Chọn một cách lựa chọn các đồ vật cho vào túi sao cho trọng lượng không quá M và tổng giá trị là lớn nhất. Mỗi đồ vật hoặc là lấy đi hoặc là bỏ lại.

01.07.02 Mô hình hóa bài toán

01.07.02.1 Input:

- 1 số nguyên n là số lượng đồ vật $(0 \leq n \leq 10)$.
- 1 mảng số nguyên w có n phần tử, w_i là trọng lượng của đồ vật thứ i. $(0 \leq p_i \leq 100, 1 \leq i \leq n)$.
- 1 mảng số nguyên p có n phần tử, p_i là giá trị của đồ vật thứ i $(0 \leq p_i \leq 100, 1 \leq i \leq n)$
- 1 số nguyên M là trọng lượng tối đa mà túi có thể đựng $(0 \leq M \leq 100)$

01.07.02.2 Output:

- 1 số nguyên v là tổng giá tri tối đa mà có thể lấy được.
- 1 mảng nhị phân s có n phần tử ($s_i = 1$ là chọn đồ vật thứ i, và ngược lại)
- Hàm mục tiêu: $maximize\ v = \sum_{1 \leq i \leq n} p_i s_i$

01.07.02.3 Constraints:

- $s_i \in \{0, 1\}$
- $\sum_{1 \leq i \leq n} w_i s_i \leq M$

01.07.03 Thiết kế thuật toán

01.07.03.1 Phân tích đặc trưng optimal substructure, overlapping subproblems

Gọi $d(i, m)$ là giá trị tối đa ta có được sau khi đưa ra quyết định chọn hay không chọn đồ vật thứ i. Trong đó, m là tổng trọng lượng tối đa của túi có thể chứa được.

Mối quan hệ giữa các trạng thái:

$$d(i, m) = max\ (d(i-1, m), v_i + d(i-1, m-w_i)$$

Khi đó, ta muốn d(i,m) đạt giá trị tối ưu, thì dĩ nhiên $d(i-1, m)$, $d(i-1, m-w_i)$ phải tối ưu, một lời giải tối ưu chứa lời giải tối ưu của bài toán nhỏ hơn ⇒ đây là đặc trưng Optimal substructure. (1)

Bên cạnh đó, có thể sẽ xảy ra trường hợp như:

$$d(4,10) = max\big(d(3,10), 4 + d(3,6)\big)$$
$$d(4,6) = max\big(d(3,6), 4 + d(3,2)\big)$$

với $w_4 = 4, v_4 = 4$

Ta thấy $d(3,6)$ phải được giải lại nhiều lần ⇒ Đặc trưng Overlapping subproblems. (2)

Từ (1), (2) suy ra bài toán có thể giải bằng kỹ thuật Quy hoạch động.

01.07.03.2 Xác định phương trình quy hoạch động

Khi $i = 0$, không có vật 0 vào để chọn cả ⇒ giá trị túi là 0.
Khi $m = 0$, túi không thể chứa vật nào cả ⇒ giá trị túi là 0.
Phương trình Quy hoạch động:

$$d(i,m) = \begin{cases} 0 & nếu\ i = 0\ hoặc\ m = 0 \\ \max\big(d(i-1,m),\ v_i + d(i-1,m-w_i)\big) & còn\ lại \end{cases}$$

01.07.03.3 Tạo bảng và lưu giá trị

```
init matrix d has n + 1 rows, M + 1 columns
for i:= 0 → n do:
        for j:= 0 → M do:
                if i==0 or j==0 then:
                        d[i][j] = 0
                else if w[i] > j then:
                        d[i][j] = d[i-1][j]
                else:
                        d[i][j] = max(d[i-1][j], v[i] + d[i-1][j-w[i]])
        end
end
```

01.07.03.4 Tra bảng, xây dựng lời giải

Tổng giá trị tối đa có thể lấy được trong bài toán chính là $d(n, M)$

```
print d[n, M]
```

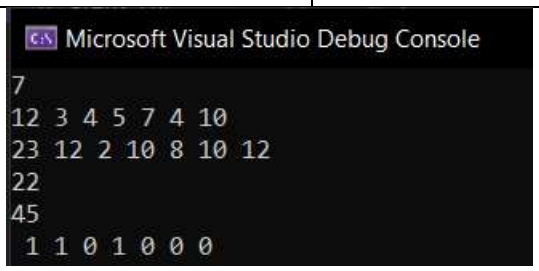
Phương án chọn đồ vật:

```
init array sol has n + 1 elements with value = 0
res = d[n][M]
m = M
for i:= n → 1 do:
        if res == 0 then:
                break;
        else if res == d[i-1][m] then:
                sol[i] = 0
```

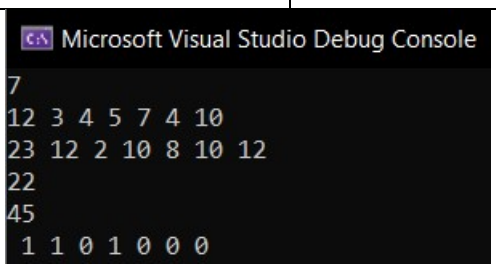
```
        else then:
                sol[i] = 1
                res -= v[i]
                m -= w[i]
end
        return sol
```

01.07.04 Ví dụ minh họa

Input	Output
5 12 2 1 4 1 4 2 1 10 2 15	15 0 1 1 1 1



Input	Output
7 12 3 4 5 7 4 10 23 12 2 10 8 10 12 22	45 1 1 0 1 0 0 0



Input	Output
10 12 3 4 9 3 7 5 7 4 10 23 12 2 12 18 17 10 8 10 12 30	80 1 1 0 0 1 1 1 0 0 0

01.07.05 Phân tích độ phức tạp

Bước tạo bảng: $\sum_{i=0}^{n}(M-0+1) = O(n.M)$
Bước xây dựng lời giải: $1+n = O(n)$
Độ phức tạp của giải thuật: $O(n.M+n) = O(n.M)$

01.08 Bài toán 6: Longest Common Subsequence

01.08.01 Mô tả bài toán

- Cho 2 dãy s_1 và s_2, tìm độ dài của dãy con chung dài nhất
  + Dãy con chung của 1 dãy có thể được tạo thành bằng cách xóa một số phần tử mà không làm thay đổi thứ tự của các phần tử còn lại.
  + Dãy con chung của hai dãy là dãy con xuất hiện trong cả 2 dãy.

01.08.02 Mô hình hóa bài toán

01.08.02.1 Input

- 2 chuỗi S_1, S_2

01.08.02.2 Output

- Số nguyên k: Độ dài chuỗi con chung dài nhất của hai dãy S_1, S_2

01.08.03 Thiết kế thuật toán

01.08.03.1 Ý tưởng giải quyết

- Gọi $S[1 \dots m]$ là chuỗi được tạo từ các phần tử của chuỗi S từ vị trí 1 đến vị trí m
- Gọi $LCS(S_1, S_2)$ là độ dài chuỗi con chung dài nhất của hai chuỗi S_1, S_2
- Đầu vào bài toán:
  + Chuỗi S_1 kích thước m

+ Chuỗi S_2 kích thước n

– Ý tưởng giải quyết: So kí tự cuối cùng trong hai dãy

+ Trường hợp 1: Kí tự cuối cùng của 2 dãy bằng nhau $S_1[m] = S_2[n]$

$$LCS(S_1[1 \dots m], S_2[1 \dots n]) = 1 + LCS(S_1[1 \dots m-1], S_2[1 \dots n-1])$$

+ Trường hợp 2: Kí tự cuối cùng của 2 dãy khác nhau $S_1[m] \neq S_2[n]$

$$LCS(S_1[1 \dots m], S_2[1 \dots n]) = \max\big(LCS(S_1[1 \dots m-1], S_2[1 \dots n]), LCS(S_1[1 \dots m], S_2[1 \dots n-1])\big)$$

01.08.03.2 Phân tích đặc trưng Optimal Substructure, Overlapping Subproblem

– Xét 2 công thức đưa ra ở mục **02.02.03.1** với 2 chuỗi $s_1[1 \dots m]$ và $s_2[1 \dots n]$

+ Trường hợp 1 – hai phần tử cuối cùng mỗi dãy bằng nhau, tiến hành loại bỏ hai kí tự cuối cùng đó khỏi mỗi chuỗi. Khi đó độ dài dãy con chung của 2 dãy $S_1[1 \dots m]$ và $S_2[1 \dots n]$ đạt giá trị lớn nhất thì độ dài dãy con chung của 2 dãy $S_1[1 \dots m-1]$ và $S_2[1 \dots n-1]$ cũng phải lớn nhất.

+ Trường hợp 2 – hai phần tử cuối cùng mỗi dãy khác nhau, chia trường hợp thành hai giai đoạn. Giai đoạn 1, loại bỏ kí tự cuối cùng $S_1[m]$ ra khỏi chuỗi $S_1[1 \dots m]$, tìm độ dài dãy con chung dài nhất của chuỗi $S_1[1 \dots m-1]$ với chuỗi $S_2[1 \dots n]$. Giai đoạn 2, loại bỏ kí tự cuối cùng $S_2[n]$ ra khỏi chuỗi $S_2[1 \dots n]$, tìm độ dài dãy con chung dài nhất của chuỗi $S_2[1 \dots n-1]$ với chuỗi $S_1[1 \dots m]$. Khi đó độ dài dãy con chung của 2 dãy $S_1[1 \dots m]$ và $S_2[1 \dots n]$ đạt giá trị lớn nhất thì độ dài dãy con chung khi loại bỏ m khỏi chuỗi $S_1[1 \dots m]$ hoặc n khỏi chuỗi $S_2[1 \dots n]$ phải lớn nhất.

+ Từ phân tích trên ⟹ Đây là đặc trưng Optimal Substructure

– Dựa vào 2 công thức đưa ra ở **01.08.03.1**, triển khai ý tưởng có thể sử dụng đệ quy với ý tưởng:

```
function LCSRecur(s1, s2, i, j)
     if i == 0 or j == 0
          return 0
     else if s1[i] == s2[j]
          return 1 + LCSRecur(s1, s2, i-1, j-1)
     else
          return max (
                         LCSRecur(s1, s2, i-1, j),
                         LCSRecur(s1, s2, i, j-1)
          )
```

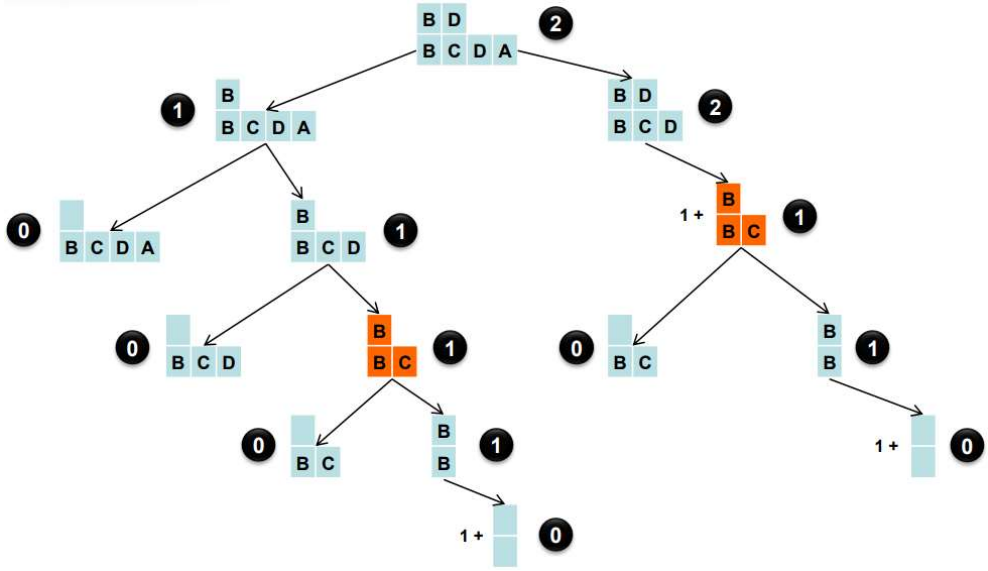
+ Trong đó i là kích thước S_1, j là kích thước S_2

+ $i = 0$ hoặc $j = 0$, một trong hai dãy rỗng → dừng thuật toán

– Minh họa ví dụ

+ Cho dãy S_1: BD

+ Cho dãy S_2: $BCDA$
+ Tìm $LCS(S_1, S_2)$
+ Xây dựng cây đệ quy



  - Trong đó, chấm đen thể hiện độ dài dãy con chung dài nhất của 2 dãy đang xét
  - Dựa trên hình, $LCS(BBC)$ (màu cam) được tính 2 lần
+ Nếu tăng chiều sâu của cây đệ quy, có thể có rất nhiều bài toán con được lặp lại không cần thiết $\Longrightarrow$ Đây là đặc trưng Overlapping Subproblem

– Có thể giải bài toán bằng quy hoạch động

01.08.03.3 Xác định phương trình quy hoạch động

$$LCS(A_i, B_j) = \begin{cases} 0 & i = 0 \; or \; j = 0 \\ LCS(A_{i-1}, B_{j-1}) + 1 & A[i] = B[j] \\ \max\left(LCS(A_i, B_{j-1}), LCS(A_{i-1}, B_j)\right) & A[i] \neq B[j] \end{cases}$$

– Trong đó
  + $LCS(A, B)$ là độ dài dãy con chung dài nhất của 2 dãy A và B
  + A_i là dãy được tạo từ dãy A bằng cách giữ lại các phần tử từ $1 \rightarrow i$ trong dãy A
  + $A[i]$ là phần tử thứ i của dãy A
– Trường hợp $LCS(A_i, B_j) = 0$ khi $i = 0$ hoặc $j = 0$ là bài toán cơ sở của vấn đề

01.08.03.4 Tạo bảng và lưu giá trị

– Cho 2 chuỗi đầu vào A, B có kích thước lần lượt n, m
– Khởi tạo bảng lưu trữ giá trị có kích thước $(n + 1) \times (m + 1)$ hoặc $(m + 1) \times (n + 1)$, bảng lưu trữ là một mảng hai chiều D
– Dựa vào phương trình quy hoạch động ở **02.02.03.3**, xây dựng và lưu trữ kết quả như sau:
  + Nếu $A[i] = B[j] \rightarrow D[i][j] = D[i - 1][j - 1]$

+ Nếu $A[i] \neq B[j] \rightarrow D[i][j] = \max(D[i-1][j], D[i][j-1])$

```
function create_table_LCS (a, b)
     n = size(a)
     m = size(b)
     D[n + 1][m + 1]
     for i = 0 -> n
          for j = 0 -> m
               if i == 0 or j == 0
                    D[i][j] = 0
               else if a[i] == b[j]
                    D[i][j] = D[i - 1][j - 1] + 1
               else
                    D[i][j] = max(D[i - 1][j], D[i][j - 1])
     return D
```

01.08.03.5 Tra bảng và xây dựng lời giải

- Xét bảng được tạo ở **01.08.03.4**, mỗi ô $D[i][j]$ trong bảng thể hiện $LCS(A_i, B_j)$ với A_i là dãy được tạo từ dãy A bằng cách giữ lại các phần tử từ $1 \rightarrow i$ trong dãy A.
- Bài toán cần tính $LCS(A, B)$, tức là $LCS(A_n, B_m)$ với n là kích thước dãy A và m là kích thước B. Theo bảng lưu trữ, $LCS(A_n, B_m)$ là kết quả được lưu trữ ở ô $D[n][m]$, do đó, ngay sau giai đoạn tạo bảng lưu trữ có thể trả về ngay kết quả ô $D[n][m]$ và đây là kết quả cuối cùng cần tìm.

```
function create_table_LCS (a, b)
     n = size(a)
     m = size(b)
     D[n + 1][m + 1]
     for i = 0 -> n
          for j = 0 -> m
               if i == 0 or j == 0
                    D[i][j] = 0
               else if a[i] == b[j]
                    D[i][j] = D[i - 1][j - 1] + 1
               else
                    D[i][j] = max(D[i - 1][j], D[i][j - 1])
     return D[n][m]
```

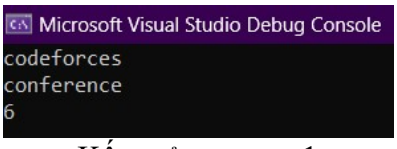
01.08.04 Ví dụ minh họa

Input
codeforces conference

Input
aaaaaaaaaaa abababab

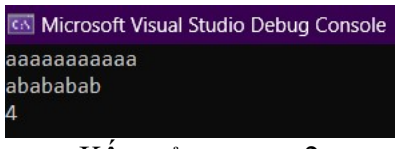
Input
lcslcslcslcs sclsclsclscl

Testcase 1



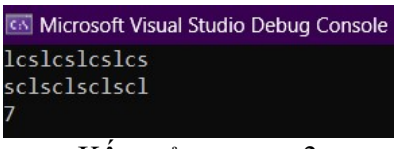
Kết quả testcase 1

Testcase 2



Kết quả testcase 2

Testcase 3



Kết quả testcase 3

01.08.05 Truy xuất giá trị

- Longest Common Subsequence cho biết độ dài dãy con chung dài nhất xuất hiện trong hai chuỗi. Thực hiện truy xuất ngược từ bảng lưu trữ để truy xuất các phần tử xuất hiện trong dãy con chung theo quy tắc sau:
  + Khởi tạo $i = n$ là dòng cuối cùng của bảng, $j = m$ là cột cuối cùng của bảng, $D[i][j]$ là ô nằm góc phải dưới của bảng (giá trị ô $D[i][j]$ là kết quả của bài toán)
  + Khởi tạo stack rỗng
  + Khởi tạo vòng lặp với điều kiện lặp $i \neq 0$ và $j \neq 0$
    - Nếu $D[i-1][j] = D[i][j]$ (1): Nhảy lên ô trên
    - Nếu $D[i][j-1] = D[i][j]$ (2): Nhảy qua ô bên trái
      ** Điều kiện (2) chỉ xảy ra khi điều kiện (1) không đúng*
    - Nếu không thỏa cả điều kiện (1) và (2), hay $D[i][j] = D[i-1][j-1]$: Thêm phần tử $A[i]$ (hoặc $B[j]$) vào stack. Nhảy lên ô chéo trên bên trái.
  + Kết thúc vòng lặp, chuỗi giá trị từ trên xuống trong stack là chuỗi con chung dài nhất của 2 chuỗi A và B.

```
i = n
j = m
init stack
while i != 0 and j != 0
     if D[i][j] == D[i-1][j]
          i--
     else if D[i][j] == D[i][j-1]
          j--
     else
          i--
          j--
          stack push A[i] or push B[j]
print stack
```

01.08.06 Phân tích độ phức tạp

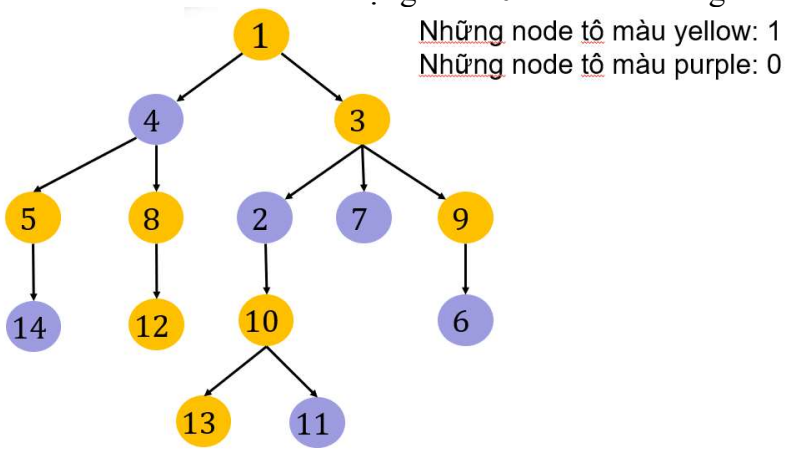
- 2 chuỗi đầu vào có kích thước lần lượt n, m
- Bảng lưu trữ được biểu diễn dưới dạng mảng 2 chiều có kích thước $n \times m$
- Duyệt toàn bộ mảng hai chiều để tính giá trị mỗi ô tương ứng
- Độ phức tạp thuật toán: $O(n \times m)$

01.09 Bài toán 7: Dynamic Programming on Tree

Dynamic Programming on Tree (DP on tree)−Quy hoạch động trên cây được dùng để giải quyết các bài toán trên cây, dựa vào mối quan hệ giá trị giữa các node để xác định kết quả của bài toán của mỗi node hoặc toàn bộ cây

01.09.01 Mô tả bài toán

Cho cây T có N node với node gốc là 1, mỗi một node có giá trị nhị phân {0,1}. Hãy tìm số lượng các subtree T' mà ở đó số lượng node 0 và node 1 bằng nhau.



+ Kết quả là những subtree 2,9,5,3

01.09.02 Mô hình hóa bài toán

01.09.02.1 Input:

- Dòng đầu tiên chứa số nguyên dương n, số lượng các node
- $n-1$ dòng tiếp theo chứa 2 số nguyên dương u, v là cạnh nối đỉnh u và v
- Dòng tiếp theo chứa n số nguyên {0,1} là màu của các node

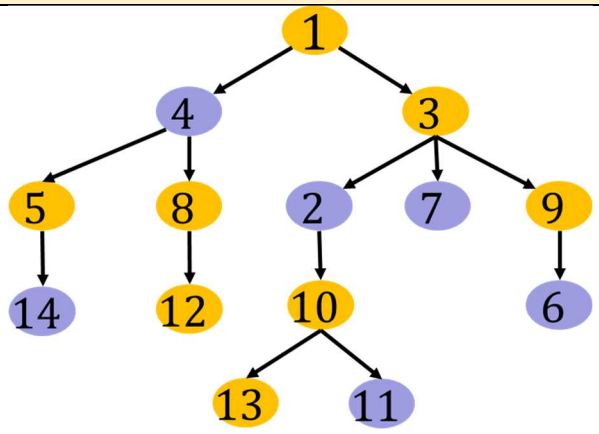
01.09.02.2 Output:

- Các subtree số lượng màu bằng nhau

01.09.03 Thiết kế thuật toán

- Cách làm Naïve:
  + Mỗi lần duyệt đến một đỉnh u nào đó trên cây T
    - Đếm các node 0 và node 1 trên các node $v \in subtree\ u$ tính cả node gốc
    - Nếu số lượng node 0 bằng số lượng node 1 thì u là valid subtree root u
  + Độ phức tạp: $O(V \times (V + E)) = O(V \times (V + V - 1)) = O(V^2)$
- Mã giả Naïve:

```
cntone=cntzero=0
def countSubTree(u):
        if color[u]==1:
                cntone++
        else:
                cntzero++
        for v ∈ child[u]:
                countSubTree(v)

def listValidSubTree(T):
        for u ∈ T:
                cntone=cntzero=0
                countSubTree(u)
                if cntone==cntzero:
                        u is valid subtree
```



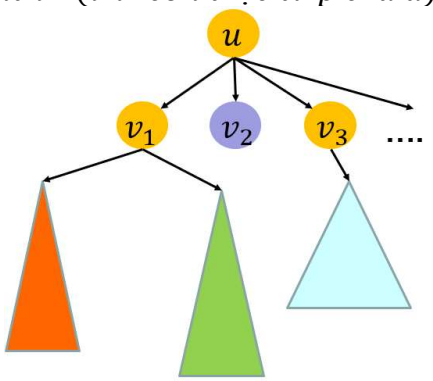
  + Xử lí node 1:
    - 1 4 3 5 8 2 7 9 14 12 10 6 13 11
  + Xử lí node 4:
    - 4 5 8 14 12
  + Xử lí node 3:
    - 3 2 7 9 10 13 11 6

+ ….

Overlapping Subproblem

- Tổng quát hóa, giả sử ta xét node u:
  + $v_1, v_2, v_3, ..$ là các node con của node u
  + Kết quả của node u sẽ là:

$node\ (u) + \sum result\ v \quad (v\ là\ con\ trực\ tiếp\ của\ u)$



- Đây là kết quả của **optimal substructure**. Kết quả tối ưu của node u được cấu thành kết quả tối ưu của các node con trực tiếp của u. Kết quả của các node cha của u không ảnh hưởng đến các $node\ \in subtree\ u$
- Gọi $d[i][j]$ là số lượng node có màu là j trong subtree gốc i
- Công thức truy hồi:

$$d[u][j] = d[u][j] + \sum_{j=0}^{color} \sum_{v \in st_u} d[v][j]$$

- Bài toán cơ sở:

$$d[u][color[u]] = 1\ ;\ \forall\ u \in T$$

- Hiện thức hóa bằng phương pháp duyệt theo chiều sâu DFS. Vì kết quả của node gốc được cấu thành bằng việc kết hợp kết quả của những node con trực tiếp. (Bước quay lui DFS). Để biết kết quả của node u thì phải biết kết quả của các node con v_i
- Mã giả:

```
d[][]
def calcSubtree(u):
        for v ∈ child[u]:
                calcSubtree(v)
                for c in color:
                        d[u][c]+=d[v][c]
def listValidSubTree(T):
        root=1
        calcSubTree(root)
        for u ∈ T:
                if d[u][0]==d[u][1]:
                        u is valid subtree
```

01.09.04 Ví dụ minh họa

– Testcase1:

Input	Output
14 1 4 1 3 4 5 4 8 3 2 3 7 3 9 5 14 8 12 2 10 9 6 10 13 10 11 1 0 1 0 1 0 0 1 1 1 0 1 1 0	2 3 5 9

+ Giải thích bảng phương án d

d	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	6	2	4	2	1	1	1	0	1	1	1	0	0	1
1	8	2	4	3	1	0	0	2	1	2	0	1	1	0

```
14
1 4
1 3
4 5
4 8
3 2
3 7
3 9
5 14
8 12
2 10
9 6
10 13
10 11
1 0 1 0 1 0 0 1 1 1 0 1 1 0
6 2 4 2 1 1 1 0 1 1 1 0 0 1
8 2 4 3 1 0 0 2 1 2 0 1 1 0

D:\SourceCode\SourceMVS\CompetitiveProgramming\Debug\dpsubtree.exe (process 21196) exited with code 0.
Press any key to close this window . . .
```

01.09.05 Phân tích độ phức tạp

– Độ phức tạp của thuật toán sẽ là $O(n)$ duyệt cây DFS

– Mục đích của bài này là giúp chúng ta mô hình hóa được kĩ thuật QHĐ được sử dụng trên cây như thế nào

- Các bài tập mở rộng nâng cao:
  + Knapsack on Tree
  + Tìm đường đi ngắn nhất giữa 2 đỉnh bất kì trên cây
  + Tìm số đường đi có độ dài đúng bằng k trên cây
  + Kết hợp với các cấu trúc dữ liệu nâng cao, bài toán truy vấn trên cây

01.10 Bài toán 8: Sparse Table

Sparse Table là một cấu trúc dữ liệu mạnh mẽ trong việc xử lý các bài toán truy vấn khoảng (range query). Cấu trúc dữ liệu này có thể giải quyết hầu hết các bài toán truy vấn khoảng với độ phức tạp $O(\log(n))$, nhưng với các bài toán dạng truy vấn khoảng có tính chất kết hợp (như tìm giá trị nhỏ nhất) thì cấu trúc dữ liệu này cho phép truy vấn chỉ với $O(1)$.

Trong phạm vi bài báo cáo này chúng em sẽ chỉ giới thiệu Sparse Table thông qua việc giải quyết bài toán nhiều truy vấn giá trị nhỏ nhất trong khoảng để có thể phát huy tối đa sức mạnh của cấu trúc dữ liệu này.

01.10.01 Mô tả bài toán

Cho dãy các số nguyên $A = a_0, a_1, a_2, \dots, a_{n-1}$ gồm n phần tử. Cho q khoảng truy vấn gồm hai số nguyên l và r với $(0 \le l \le r < n)$. Với mỗi khoảng truy vấn, hãy cho biết giá trị nhỏ nhất của các phần tử trong mảng A trong đoạn $[l, r]$.

01.10.02 Mô hình hóa bài toán

01.10.02.1 Input:

- Dòng thứ nhất chứa một số nguyên dương n là độ dài của dãy A đầu vào.
- Dòng thứ hai chứa n số, là các phần tử trong dãy A.
- Dòng thứ ba chứa một số nguyên dương q là số khoảng truy vấn.
- q dòng tiếp theo, mỗi dòng chứa hai số nguyên dương l và r $(0 \le l \le r < n)$

01.10.02.2 Output:

- q dòng, mỗi dòng là giá trị nhỏ nhất trong khoảng $[l, r]$ của mảng A.
- Lưu ý: số thứ tự trong mảng bắt đầu từ 0.

01.10.02.3 Ví dụ:

Input	Output
3 1 4 1 2	4 1

1 1 1 2	

– Giải thích:
  - Ở khoảng truy vấn đầu tiên, mảng A trong khoảng $[1,1]$ chỉ có một phần tử là $\{4\}$ nên giá trị nhỏ nhất của đoạn là 4.
  - Ở khoảng truy vấn thứ hai, mảng A trong khoảng $[1,2]$ có hai phần tử là $\{1, 4\}$ và giá trị nhỏ nhất của đoạn là 1.

01.10.03 Thiết kế thuật toán

Sparse Table là cấu trúc dữ liệu có dạng ma trận 2 chiều. Chỉ số cột và hàng của ma trận này biểu diễn cho độ dài của từng khoảng cần truy vấn và vị trí của khoảng truy vấn đó ở trong mảng. Độ dài của các khoảng truy vấn được lưu trong Sparse Table luôn là lũy thừa của 2. Bảng không nhất thiết phải được lấp đầy chính vì vậy nên được gọi là Sparse Table.

Với mỗi khoảng truy vấn $[l, r]$ nhận được, ta cần tìm giá trị nhỏ nhất của mảng trong đoạn đó. Thông thường, ta có thể duyệt qua lần lượt từng phần tử trong khoảng để tìm ra giá trị nhỏ nhất. Hoặc ta có thể chia khoảng $[l, r]$ thành nhiều khoảng nhỏ, tìm giá trị nhỏ nhất trong các đoạn đó rồi so sánh chúng với nhau, ta sẽ có được giá trị nhỏ nhất trong đoạn $[l, r]$. Tuy nhiên, khi bài toán yêu cầu nhiều khoảng truy vấn thì việc lần lượt tìm giá trị nhỏ nhất trong từng khoảng bằng cách chia để trị là không tối ưu về mặt thời gian. Do đó, ta sẽ lưu lại các giá trị nhỏ nhất trong các khoảng đó bằng Sparse Table để phục vụ việc truy vấn dễ dàng.

Như đã Nêu, ý tưởng của Sparse Table là lưu lại giá trị của các khoảng truy vấn sao cho mỗi khoảng có độ dài là lũy thừa của 2. Với mỗi số nguyên dương bất kỳ, ta có thể phân tích thành tổng các số nguyên là lũy thừa của 2. Ví dụ:

$13 = 8 + 4 + 1 = 2^3 + 2^2 + 2^0$

$27 = 16 + 8 + 2 + 1 = 2^4 + 2^3 + 2^1 + 2^0$

Ta lại có: Với mỗi đoạn $[l, r]$ bất kỳ, độ dài của đoạn là $r - l + 1$. Do đó, ta có thể chia đoạn $[l, r]$ thành các đoạn nhỏ có độ dài là lũy thừa của 2.Ví dụ:

$[0, 26] = [0, 15] + [16, 23] + [24, 25] + [26,26]$

Do đó, ta có thể lưu lại giá trị nhỏ nhất của tất cả các đoạn con có độ dài là lũy thừa của 2 của dãy A để từ đó trả lời cho mỗi khoảng truy vấn $[l, r]$ bằng cách tìm các giá trị nhỏ nhất trong số các giá trị nhỏ nhất của các khoảng trong đoạn $[l, r]$. Cách giải này có hai tính chất:

1. Tìm được giá trị nhỏ nhất của các đoạn trong khoảng $[l, r]$ sẽ tìm được giá trị nhỏ nhất trong khoảng $[l, r]$ ⇒ **Optimal substructure**
2. Ở mỗi đoạn trong khoảng $[l, r]$ cần tìm ra giá trị nhỏ nhất trong đoạn đó ⇒**Overlapping subproblems**.

Vậy ta có thể giải cách này bằng phương pháp Quy Hoạch Động. Khi đó, vì một đoạn bất kỳ có thể chia nhỏ nhất thành đoạn có độ dài là 1, đoạn này chỉ có một phần tử

nên giá trị nhỏ nhất của đoạn cũng là chính nó. Do vậy, ta có **bài toán cơ sở** là: các đoạn có độ dài là 1 sẽ có giá trị nhỏ nhất là phần tử đó.

01.10.03.1 Xây dựng Sparse Table

Sparse Table là cấu trúc dữ liệu sử dụng ma trận hai chiều. Do vậy, trong ma trận này ta có thể coi chỉ số cột tương ứng với chỉ số của mảng (số lượng cột bằng số lượng phần tử của mảng A), chỉ vị trí bắt đầu của khoảng trong mảng. Chỉ số hàng là lũy thừa 2 là độ dài của một đoạn. Ví dụ: hàng 3 sẽ lưu giá trị nhỏ nhất của các đoạn có độ dài là $2^3 = 8$, cột 4 sẽ lưu giá trị nhỏ nhất của các đoạn có vị trí bắt đầu là 4 trong mảng. Như vậy, một ô trong Sparse Table sẽ giữ 3 thông tin:

- Độ dài của đoạn
- Vị trí bắt đầu của đoạn trong mảng
- Giá trị nhỏ nhất trong đoạn.

Mã giải quá trình xây dựng Sparse Table:

```
def SparseTableBuilding()
        Initialize 2D array spT which has |log(n)+1| rows and n column;

        Copy all array to the first row of spT

        for (expo: 1 -> |log(n)|)
        {
                for (i=0; i+2^expo<n; i++)
                spT[expo][i] = min(spT[expo-1][i],spT[expo-1][i+2^(expo-1)]);
        }
        return spT;
```

Sau khi đã có Sparse Table, ta có thể thực hiện các truy vấn. Nhận thấy rằng, với mỗi khoảng $[l, r]$, ta có thể sử dụng đệ quy để chia đoạn $[l, r]$ thành nhiều khoảng để lấy ra giá trị nhỏ nhất của từng khoảng trong bảng với độ phức tạp $O(logn)$. Tuy nhiên, ta có thể chia đoạn $[l, r]$ thành hai đoạn có độ dài bằng nhau và trùng lắp với nhau và vì phép tính giá trị nhỏ nhất có tính chất kết hợp nên ta có thể lấy vùng trùng lắp mà không làm thay đổi kết quả bài toán. Ví dụ minh họa cho đoạn $[1,7]$ sau:

$[1,7] = [1,4] + [4,7]$

Có thể thấy đoạn $[1,4]$ và đoạn $[4,7]$ có phần tử trùng là phần tử tại $index = 4$, nhưng phần trùng này sẽ không ảnh hưởng kết quả do vậy ta có thể tận dụng để thực hiện truy vấn với độ phức tạp $O(1)$ vì hai đoạn này có độ dài bằng nhau (bằng 4) do đó giá trị nhỏ nhất của hai đoạn sẽ cùng nằm trên một hàng trong bảng Sparse Table. Khi đó ta có thể thực hiện so sánh hai ô này với nhau và sẽ lấy được kết quả với độ phức tạp chỉ $O(1)$.

Tổng quát, với mỗi đoạn $[l, r]$, ta có thể thực hiện so sánh giữa hai ô trong bảng Sparse Table là ô tại vị trí hàng $\log_2 r - l + 1$ cột l với ô tại vị trí hàng $\log_2 r - l + 1$ cột $r - 2^{\log_2 r - l + 1} + 1$.

Mã giải cho mỗi truy vấn như sau:

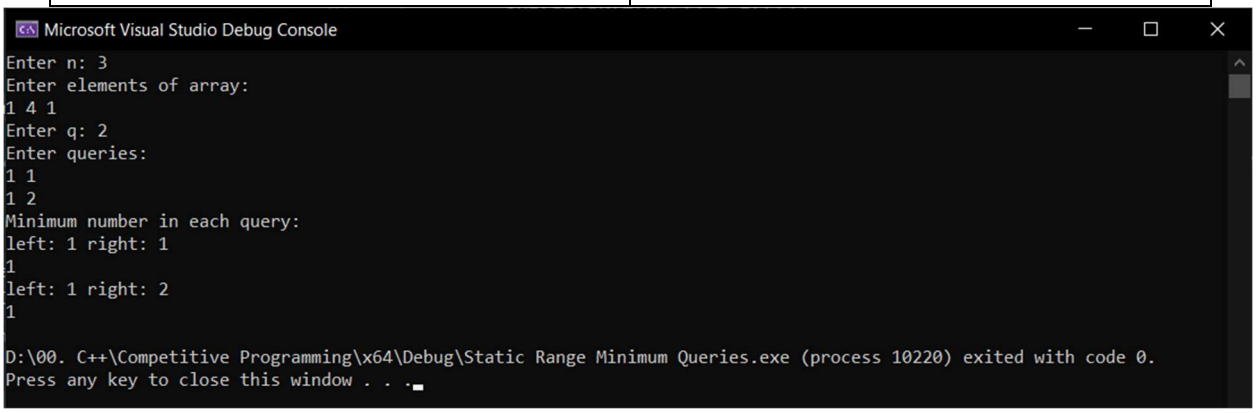
```
def SparseTableBuilding(left, right, spT)
```

```
        power = log2(right-left+1);
        return min(spT[power][left], spT[power][right-2^power + 1]);
```

01.10.04 Ví dụ minh họa

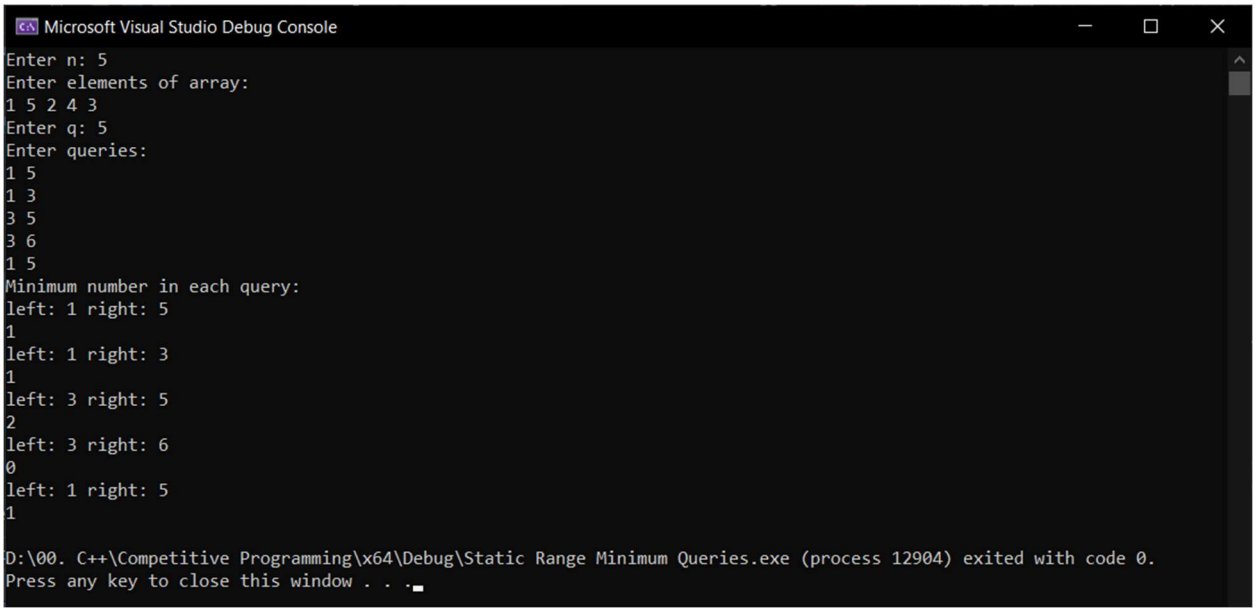
– Test case 1:

Input	Output
3 1 4 1 2 1 1 1 2	4 1



– Test case 2:

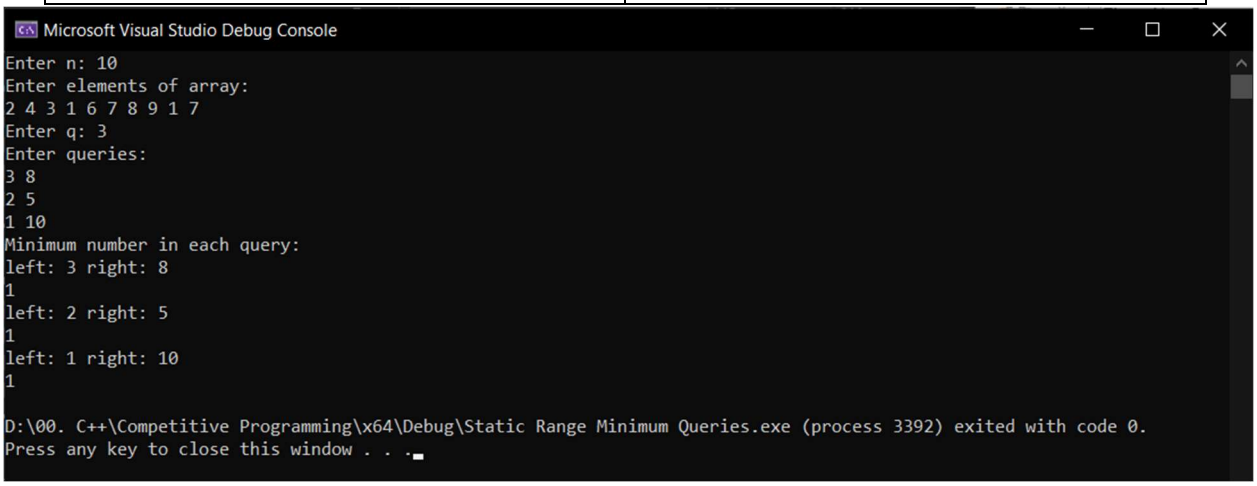
Input	Output
5 1 5 2 4 3 5	1 1 2



1 5 1 3 3 5 3 6 1 5	0 1

– Test case 3:

Input	Output
10 2 4 3 1 6 7 8 9 1 7 3 3 8 2 5 1 10	1 1 1



01.11 Bài toán 9: Matrix Exponentiation

01.11.01 Mô tả bài toán

- Cho biểu thức A, với A có thể là một con số, một đa thức, một ma trận, …, tính lũy thừa bậc k của A với k có thể là một giá trị cực lớn.
- Lũy thừa ma trận (Matrix Exponentiation) là một trong những phép lũy thừa được ứng dụng rộng rãi trong các bài toán thực tế.

01.11.02 Mô hình hóa bài toán

01.11.02.1 Input

- Số nguyên dương n: Kích thước ma trận A
- Số nguyên dương k: Số mũ của phép lũy thừa
- Ma trận A: Cơ số của phép lũy thừa

01.11.02.2 Output

- Kết quả phép tính A^k

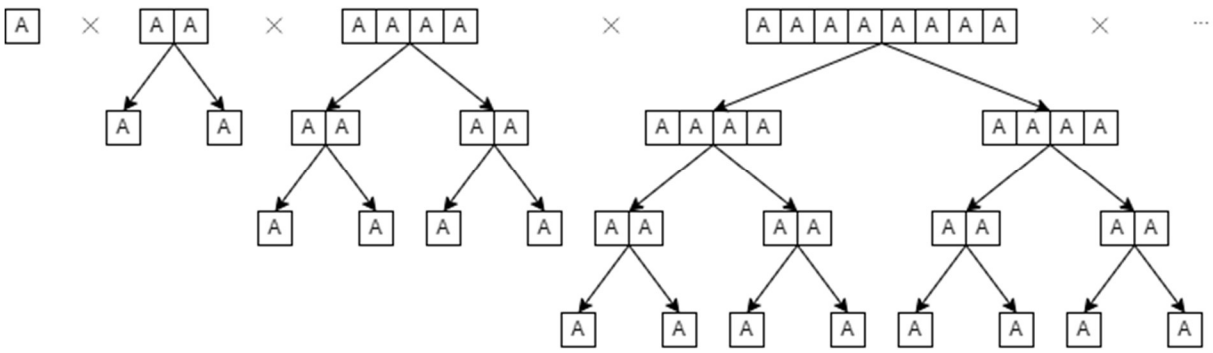
01.11.02.3 Constraint

- Kích thước ma trận A: $n \times n \rightarrow$ Ma trận vuông

01.11.03 Thiết kế thuật toán

01.11.03.1 Phân tích đặc trưng Optimal Substructure, Overlapping Subproblem

- Cho ma trận A đầu vào và số nguyên k là số mũ phép lũy thừa:
$$A^k = A \times A \times A \ldots \times A \rightarrow k \text{ thừa số } A$$
- Độ phức tạp khi nhân k thừa số A với nhau: $O(k)$ (chưa tính chi phí xử lý với ma trận)
- Ma trận có tính kết hợp, do đó phép lũy thừa có thể được biểu diễn như sau:
$$A^k = A \times (A \times A) \times (A \times A \times A \times A) \times \ldots \times \ldots$$
$$A^k = A \times A^2 \times A^4 \times \ldots \times A \times \ldots$$
  + A^k được tính bằng cách gom thành nhiều nhóm, các nhóm được biểu diễn $A^{(2^i)}$ với $1 \leq i \leq \log_2 k$
  + Công thức được biểu diễn theo mô hình dưới đây:



+ Theo mô hình trên, mỗi nhóm phải tính tích 2^i thừa số. Nếu có A
  - $A^2 = A \times A$
  - $A^4 = A \times A \times A \times A = A^2 \times A^2$
  - $A^8 = A \times A \times A \times A \times A \times A \times A \times A = A^4 \times A^4$
+ Biểu diễn mô hình trên yêu cầu tính lặp lại các bài toán nhiều lần $\Longrightarrow$ Đây là đặc trưng Overlapping Subproblem.
+ Mặc khác
  - Nếu đã có A^2, $A^4 = A^2 \times A^2$
  - Nếu đã có A^4, $A^8 = A^4 \times A^4$
  - …
+ Với biểu diễn công thức trên, để tối thiểu hóa số phép tính A^k, yêu cầu phải tối thiểu hóa số phép tính của $A^{\frac{k}{2}}$ $\Longrightarrow$ Đây là đặc trưng Optimal Substructure.

01.11.03.2 Tạo bảng và lưu giá trị

– Cho đầu vào ma trận A với số mũ k
– Tạo mảng tích lũy d chứa $\log_2 k + 1$ phần tử

$$d[0] = A$$
$$d[i] = d[i-1] \times d[i-1]$$

```
function create_table(A, k)
     n = log(k)
     init array dp[n]
     dp[0] = Identity(A.n)
     for i = 1 -> n do
          d[i] = d[i - 1] * d[i - 1]
     return d
```

01.11.03.3 Tra bảng và xây dựng lời giải

```
Matrix pow(A, d, k)
     if k == 0: return identity matrix size n
     if k == 1: return A
```

```
i = 0
ans = identity matrix size n
while k > 0
     if k is odd
          ans = ans * d[i]
     i++
     k /= 2
return ans
```

01.11.04 Ví dụ minh họa

$$\begin{pmatrix} 2 & 3 & 4 \\ 4 & 5 & 6 \\ 2 & 4 & 5 \end{pmatrix}^{8}$$

```
Microsoft Visual Studio Debug Console
3 8
2 3 4
4 5 6
2 4 5

69605836 106577157 132446706
115065600 176182993 218948016
86501310 132446706 164595589
```

$$\begin{pmatrix} 2 & 1 & 4 & 8 \\ 4 & 5 & 4 & 9 \\ 7 & 6 & 5 & 2 \\ 8 & 6 & 2 & 1 \end{pmatrix}^{11}$$

```
Microsoft Visual Studio Debug Console
4 11
2 1 4 8
4 5 4 9
7 6 5 2
8 6 2 1

16831621956164 14212018500237 11906832718030 16636522445844
25293943805762 21356405176783 17887880657954 24988028077241
23313430895741 19683012910538 16480262055129 23014565873523
19342551900648 16329918952648 13669896999114 19086499196291
```

$$\begin{pmatrix} 2 & 3 & 4 & 1 & 2 & 3 \\ 5 & 6 & 7 & 6 & 7 & 8 \\ 8 & 9 & 0 & 2 & 2 & 2 \\ 2 & 3 & 2 & 7 & 4 & 4 \\ 1 & 1 & 1 & 3 & 3 & 3 \\ 2 & 3 & 9 & 5 & 2 & 1 \end{pmatrix}^{5}$$

```
Microsoft Visual Studio Debug Console
6 5
2 3 4 1 2 3
5 6 7 6 7 8
8 9 0 2 2 2
2 3 2 7 4 4
1 1 1 3 3 3
2 3 9 5 2 1

636638 793878 715297 765355 639392 672657
1542253 1923485 1734659 1858814 1550201 1629750
1022917 1276412 1159825 1236208 1032051 1085340
846267 1055903 954251 1024613 853051 896428
431101 537701 484814 521085 433195 454807
926762 1156373 1043321 1116893 936298 987273
```

01.11.05 Phân tích độ phức tạp

- Phép lũy thừa kết hợp kỹ thuật quy hoạch động thực hiện vòng lặp $\log_2 k$ lần để trả về kết quả cuối cùng (với k là số mũ của phép lũy thừa):
  + Độ phức tạp: $O(\log(k))$
- Phép nhân hai ma trận cần 3 vòng lặp, lặp qua các dòng ma trận A, các cột ma trận B và lặp qua từng phần tử trong mỗi dòng (cột) (A, B là ma trận vuông có kích thước $n \times n$)
  + Độ phức tạp: $O(n^3)$
- Như vậy, phép ma trận lũy thừa sẽ có độ phức tạp $O(n^3 \times k)$

01.11.06 Bài toán ứng dụng

- Bài toán tìm số Fibonacci đã có giải pháp áp dụng quy hoạch động với độ phức tạp $O(n)$. Tuy nhiên, với các trường hợp n cực lớn (chẳng hạn $n = 10^{18}$), giải pháp $O(n)$ không hiệu quả. Do đó, sử dụng ma trận lũy thừa áp dụng kỹ thuật quy động giúp giải quyết vấn đề này.
- Công thức Fibonacci: $F_n = F_{n-1} + F_{n-2}$
- Biểu diễn công thức Fibonacci dưới dạng phép nhân ma trận:

$$F_n = \begin{pmatrix} 1 & 1 \end{pmatrix} \times \begin{pmatrix} F_{n-1} \\ F_{n-2} \end{pmatrix} \quad (1)$$

  + Công thức (1) sử dụng ma trận không vuông, do đó chưa thể áp dụng ma trận lũy thừa
  + Biểu diễn khác:

$$\begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \times \begin{pmatrix} F_{n-1} \\ F_{n-2} \end{pmatrix}$$

- Ta có:
  + $\begin{pmatrix} F_2 \\ F_1 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \times \begin{pmatrix} F_1 \\ F_0 \end{pmatrix}$
  + $\begin{pmatrix} F_3 \\ F_2 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \times \begin{pmatrix} F_2 \\ F_1 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \times \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \times \begin{pmatrix} F_1 \\ F_0 \end{pmatrix}$
  + $\begin{pmatrix} F_4 \\ F_3 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \times \begin{pmatrix} F_3 \\ F_2 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \times \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \times \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \times \begin{pmatrix} F_1 \\ F_0 \end{pmatrix}$
  + …
  + $\begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \times \begin{pmatrix} F_{n-1} \\ F_{n-2} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1} \begin{pmatrix} F_1 \\ F_0 \end{pmatrix}$
  + $\begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \times \begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n} \begin{pmatrix} F_1 \\ F_0 \end{pmatrix}$
  + $\begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n} \begin{pmatrix} F_1 \\ F_0 \end{pmatrix}$
- Vậy công thức Fibonacci đã biểu diễn được dưới dạng lũy thừa của ma trận $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$. Như vậy đã có thể tính số Fibonacci với độ phức tạp $O(\log(n))$
- Minh họa

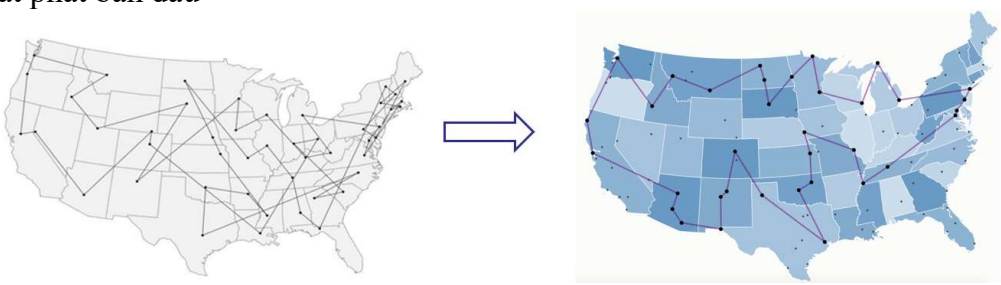
```
Microsoft Visual Studio Debug Console
2 10
1 1
1 0

55
34

F(10) = 55
```

```
Microsoft Visual Studio Debug Console
2 14
1 1
1 0

377
233

F(14) = 377
```

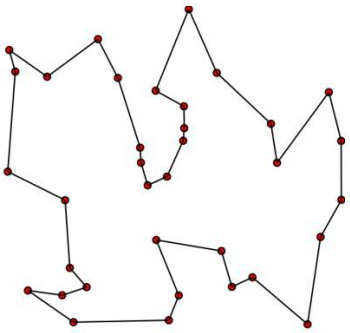
01.12 Bài toán 10: Dynamic Programming with Bitmask

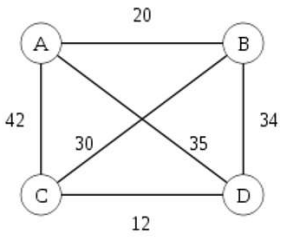
01.12.01 Mô tả bài toán

Cho một tập các thành phố và khoảng cách giữa các thành phố với nhau, tìm một đường đi ngắn nhất sao cho đi qua mỗi thành phố đúng một lần và cuối cùng về đúng vị trí xuất phát ban đầu



- Mô hình hóa: Đồ thị vô hướng || có hướng $G(V, E)$
- Mục tiêu: Tìm chu trình Hamilton ngắn nhất





01.12.02 Mô hình hóa bài toán

- Input:

+ Dòng đầu tiên chứa 2 số nguyên dương n và m là số đỉnh và số cạnh của đồ thị
+ m dòng tiếp theo chứa 3 số nguyên $u\ v\ w$ là cạnh $u\ v$ và trọng số w của đồ thị $(0 < u, v < n)$

– Output:
+ Độ dài của chu trình Hamilton ngắn nhất

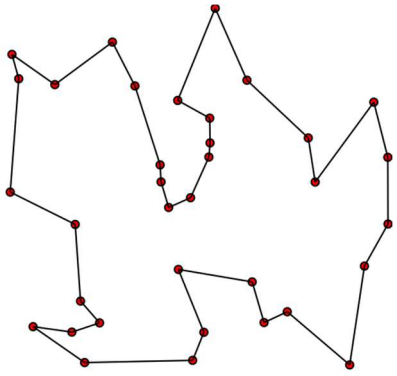
Input	Output
4 6 0 1 20 0 2 42 0 3 35 1 2 30 1 3 34 2 3 12	97

01.12.03 Thiết kế thuật toán

– Cấu hình là tập các phần tử thường được biểu diễn dưới dạng nhị phân.
– Ví dụ: Balo có 10 món đồ, nếu chọn hoặc không chọn các món đồ đó thì tổng số cấu hình: 2^{10} $(0 \rightarrow 2^{10} - 1)$

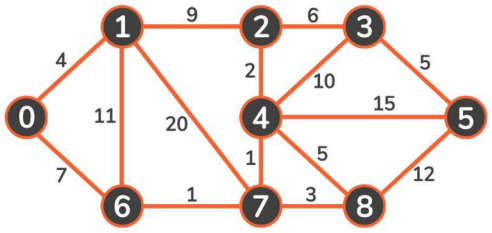
0	1	2	3	4	5	6	7	8	9
0	1	0	1	1	0	0	1	1	0

– Xét một đường đi bất kì, mỗi đỉnh trong đồ thị có thể thuộc đường đi hoặc không
– Với hai đường đi có cùng tập đỉnh (cùng cấu hình) và cùng đỉnh kết thực, khả năng mở rộng của chúng là như nhau. Trong các đường đi có cùng cấu hình và cùng điểm kết thúc, ta chỉ quan tâm các đường đi có chi phí nhỏ nhất:
+ 1 0 2
+ 0 1 2



– Để ý rằng chúng ta phân biệt những cấu hình giống nhau bằng đỉnh kết thúc
– Quy hoạch:
+ Cấu hình
+ Đỉnh kết thúc

– Gọi $d(s,u)$ là độ dài đường đi ngắn nhất kết thúc tại u và đi qua tất cả các đỉnh thuộc cấu hình s
  + Ta sẽ biểu diễn cấu hình s dưới dạng nhị phân
– Ví dụ:



  + $d(7,2) = 13$
    • $7_{10} = 000000111_2$
  + $d(2,2) = INF$
    • $2_{10} = 000000010_2$
– Có nhiều cấu hình giống nhau, chúng được phân biệt với nhau bằng đỉnh kết thực
Overlapping subproblem
– Công thức truy hồi:

$$d(s,u) = \min_{v}[d(s\backslash u,v) + w(u,v)]$$

– Bài toán cơ sở: Do chu trình luôn bắt đầu từ 0 nên

$$d[1,0] = 0$$

– Ví dụ:
  + Xét cấu hình 0 1 2 4 6 7 kết thúc tại 4
    • Cấu hình 0 1 2 6 7 kết thúc tại 2
    • Cấu hình 0 1 2 6 7 kết thúc tại 7
  + Min(W(2,4) cấu hình A và W(7,4) cấu hình B)

Optimal Substructure

– Mã giả Top-down:

```
def solve(s,u):
        if d[s][u] is defined
                return d[s][u]
        for v can reach u:
                mask = s & ~ (1<<v)
                if (mask & (1<<v)) != 0:
                        d[s][u]=min(d[s][u],solve(mask,v)+w(v,u))
def ans():
        mxmsk=(1<<n)-1
        for u = 0 -> n - 1:
                ans=min(ans,solve(mxmsk,u)+w[u][0])
        return ans
```

– Mã giả Bottom-up:

```
def solve(s,u):
        mxmsk=(1<<n)-1
```

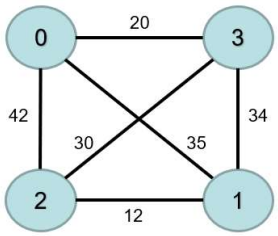
```
for s = 0 -> mxmsk:
    for u = 0 -> n - 1:
        if ((1<<u) & s) == 0:
            continue
        for v = 0 -> n - 1:
            if ((1<<v) & s) != 0 :
                continue
            msk=s|(1<<v)
            d[msk][v]=min(d[msk][v],d[s][u]+w[u][v])
```

01.12.04 Ví dụ minh họa

– Testcase1:

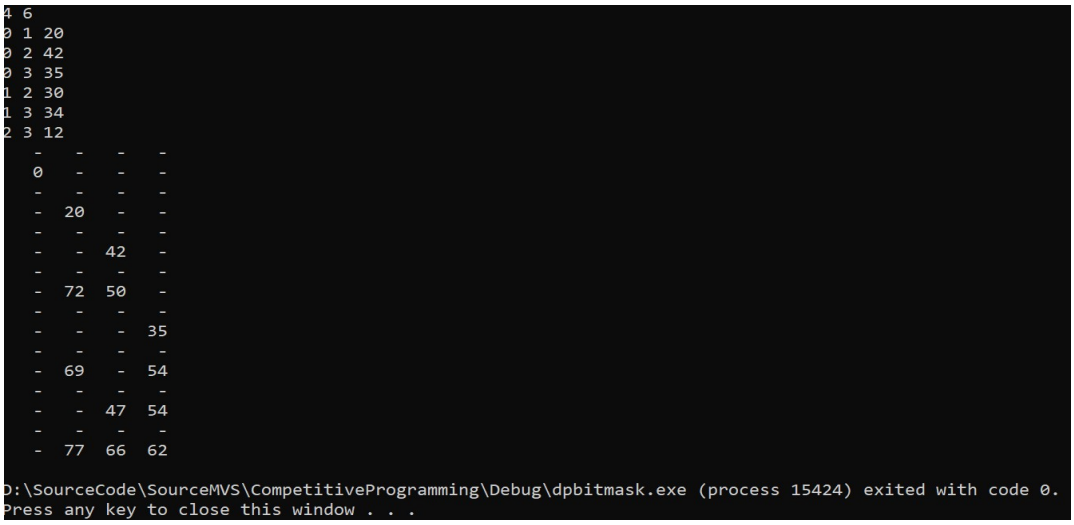
Input	Output
4 6 0 3 20 0 1 35 0 2 42 3 1 34 3 2 30 2 1 12	97

+ Giải thích



- Bảng phương án:

s\u	0	1	2	3
0				
1	0	INF	INF	INF
2	INF	INF	INF	INF
3	INF	35	INF	INF
4	INF	INF	INF	INF
5	INF	INF	42	INF
6	INF	INF	INF	INF
7	INF	54	47	INF
8	INF	INF	INF	INF
9	INF	INF	INF	20
10	INF	INF	INF	INF
11	INF	54	INF	69
12	INF	INF	INF	INF
13	INF	INF	50	72
14	INF	INF	INF	INF
15	INF	62	66	77



01.12.05 Phân tích độ phức tạp

- Tổng số lượng cấu hình là : 2^n. Với mỗi cấu hình lặp n^2 cặp đỉnh u, v.
- Độ phức tạp cảu thuật toán là $O(2^n n^2)$

01.13 Bài toán 11: Knuth's optimization

01.13.01 Knuth's optimization

01.13.01.1 Khái niệm

Knuth's optimization (Knuth-Yao speedup), là một trường hợp đặc biệt của quy hoạch động trên các dãy, có thể tối ưu hóa độ phức tạp về thời gian của các giải pháp theo hệ số tuyến tính, từ $O(n^3)$ thành $O(n^2)$.

01.13.01.2 Điều kiện áp dụng

- Giải pháp quy hoạch động có dạng:
$$d(i, j) = \min_{i \le k < j} (d(i, k) + d(k+1, j) + C(i, j))$$
- Hàm chi phí C thỏa mãn điều kiện:
$$C(b, c) \le C(a, d)$$
$$C(a, c) + C(b, d) \le C(a, d) + C(b, c)$$ (The quadrangle inequality [QI])
với $a \le b \le c \le d$

01.13.01.3 Hệ quả:

Gọi $opt(i,j)$ là giá trị của k mà tối thiểu hóa $d(i,j)$. Khi đó, Kỹ thuật tối ưu Knuth sẽ giới hạn miền tìm kiếm $opt(i,j)$. Từ $i \le opt(i,j) \le j-1$ sang $opt(i,j-1) \le opt(i,j) \le opt(i+1,j)$

01.13.02 Mô tả bài toán ứng dụng (Cutting Sticks)

Bạn phải cắt cây gậy gỗ thành từng khúc. Công ty nhận việc cắt gỗ sẽ tính tiền theo chiều dài của cây gậy được cắt.

Dễ dàng ta nhận thấy rằng các lựa chọn khác nhau theo thứ tự cắt có thể dẫn đến chi phí khác nhau. Ví dụ, xem xét một thanh có chiều dài $10m$ phải được cắt ở cách vạch $2m$, $4m$, $7m$ tính từ một phía. Có một số lựa chọn. Người ta có thể cắt đầu tiên vào vạch 2, sau đó 4 và cuối cùng là 7. Điều này dẫn đến chi phí là $10+8+6=24$, vì thanh đầu tiên là $10m$, sau đó là $8m$ và cuối cùng là $6m$. Lựa chọn khác có thể là cắt ở vạch 4, sau đó là 2, cuối cùng là 7. Điều này dẫn đến chi phí là $10+4+6=20$, đây là một phương án tốt hơn.

Bạn hãy tìm ra chi phí tối thiểu đề cắt một cây gậy gỗ được giao.

01.13.03 Mô hình hóa bài toán

01.13.03.1 Input:

- 1 số nguyên dương l là độ dài của gậy gỗ cần cắt. $(0 < l < 1000)$
- 1 số nguyên n là số vạch cần cắt. $(0 \le n < 50)$
- 1 mảng số nguyên dương c có n phần tử, mỗi phần tử c_i là vị trí trên gậy gỗ cần phải cắt, các phần tử được sắp xếp theo thứ tự tăng dần. $(1 \le i \le n, 0 < c_i < l)$

01.13.03.2 Output:

- 1 số nguyên là chi phí tối thiểu để cắt gậy gỗ.

01.13.04 Thiết kế thuật toán

01.13.04.1 Phân tích đặc trưng optimal substructure, overlapping subproblems

Bất cứ khi nào chúng ta thực hiện một lần cắt, chúng ta sẽ chia cây gậy thành 2 cây gậy khác. Sau đó 2 cây gậy đó sẽ cần được cắt lại (nếu có vạch cắt trên đó) hoặc sẽ giữ nguyên (nếu không còn vạch cắt nào trên đó nữa). Lưu ý rằng 2 cây gậy mới cần cắt đại diện cho cùng một vấn đề (kích thước mới, số vạch cắt mới). Với ý tưởng đó, để tìm cách tốt nhất cắt cây gậy hiện tại, chúng ta cần tìm cách tốt nhất để cắt 2 cây gậy phụ mới đó và tính tổng chi phí của chúng với chi phí của lần cắt ban đầu mà ta đã thực hiện.

01.13.04.2 Xác định phương trình quy hoạch động

Gọi $d(i,j)$ là chi phí tối thiểu để cắt cây gậy có độ dài bắt đầu từ vạch thứ i đến vạch thứ j.

Để lời giải bài toán cuối cùng cũng được trình bày giống bài toán con, ta thêm vào đầu và cuối mảng vạch cắt 2 phần tử lần lượt là $c_0 = 0$ và $c_{n+1} = l$. Có nghĩa là mảng c giờ có $n+1$ phần tử và bài toán là tìm $d(0, n+1)$.

Phương trình quy hoạch động:

$$d(i,j) = \min_{i<k<j}\big(d(i,\ k) + d(k,\ j)\big) + cost(i,j)$$

Trong đó $cost(i,j)$ là chi phí cắt cây gậy có độ dài bắt đầu từ vạch thứ i đến vạch thứ j.

$$cost(i,j) = c_j - c_i$$

01.13.04.3 Tạo bảng và lưu giá trị

```
insert 0 to head of array c
insert l to tail of array c

init matrix d with n + 2 rows, n + 2 columns
for i:= n → 0 do:
        for j:= i+1 → n+1 do:
                if  j == i+1 then:
                        d[i][j] = 0
                else then:
                        for k:= i+1 → j − 1 do:
                                d[i][j] = min(d[i][j], d[i][k]+d[k][j]+c[j]-c[i])
                        end
        end
end
```

01.13.04.4 Tra bảng, xây dựng lời giải ban đầu

```
print d[0][n+1]
```

01.13.04.5 Áp dụng Knuth's optimization

Kiểm tra điều kiện áp dụng:

- Xét giải pháp quy hoạch động:

$$d(i,j) = \min_{i<k<j}\big(d(i,\ k) + d(k,\ j)\big) + cost(i,j)$$

  thỏa mãn điều kiện
- Xét hàm chi phí: $cost(i,j) = c_j - c_i$

Với 4 số $a \le b \le c \le d \Rightarrow c_a \le c_b \le c_c \le c_d$ (mảng c tăng dần)

Ta có: $c_a \le c_b$

$c_c \le c_d$

$\Rightarrow c_a + c_c \le c_b + c_d$

$\Rightarrow c_c - c_b \le c_d - c_a$

$\Rightarrow cost(b,c) \le cost(d,a)$ (1)

Mặt khác:

$$cost(a,c) + cost(b,d) = c_c - c_a + c_d - c_b = (c_d - c_a) + (c_c - c_b)$$
$$= cost(a,d) + cost(b,c) \quad (2)$$

Từ (1) và (2) suy ra hàm chi phí thỏa điều kiện của Knuth

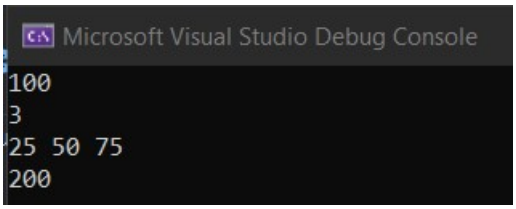
Vậy Giải pháp của ta có thể áp dụng kỹ thuật tối ưu Knuth.

Thay vì tìm giá trị optimal trong khoảng từ $i + 1$ đến $j - 1$, ta tìm trong đoạn từ $opt(i, j-1)$ đến $opt(i+1, j)$.

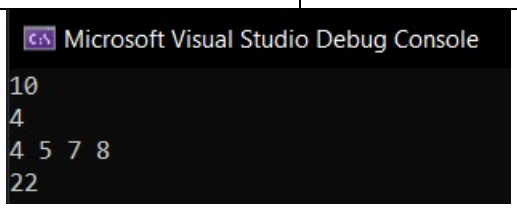
```
insert 0 to head of array c
insert l to tail of array c

init matrix d with n + 2 rows, n + 2 columns
init matrix opt with n + 2 rows, n + 2 columns
for i:= n → 0 do:
        for j:= i+1 → n+1 do:
                if  j == i+1 then:
                        d[i][j] = 0
                        opt[i][j] = j
                else then:
                        for k:= opt[i][j-1] → min(opt[i+1][j], j-1) do:
                                if d[i][k]+d[k][j]+c[j]-c[i] < d[i][j] then:
                                        d[i][j] = d[i][k]+d[k][j]+c[j]-c[i]
                                        opt[i][j] = k
                        end
        end
end
```

01.13.05 Ví dụ minh họa

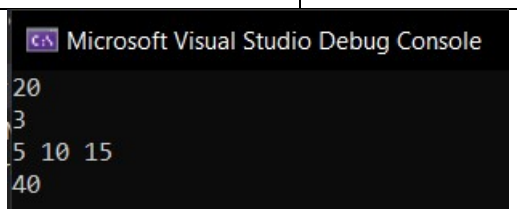
Input	Output
100 3 25 50 75	200



Input	Output
10 4 4 5 7 8	22



Input	Output
20 3 5 10 15	40



01.13.06 Phân tích độ phức tạp

Giải pháp trên có độ phức tạp:

$$\sum_{i=0}^{n}\sum_{j=i+1}^{n+1}\big(opt(i+1,j)-opt(i,j-1)\big)$$
$$=\sum_{i=0}^{n}\sum_{j=i+1}^{n+1}\big(opt(i+1,j)-opt(i,j-1)\big)$$
$$=\sum_{k=1}^{n+1}\big(opt(k,n+1)-opt(1,k)\big)$$
$$=O(n^2)$$

01.13.07 Chứng minh tính đúng đắn của Knuth's optimization

Chứng minh tính đúng đắn của Knuth's optimization có nghĩa là chứng minh:

$$opt(i,j-1)\le opt(i,j)\le opt(i+1,j)$$

khi giải pháp đã thỏa mãn điều kiện áp dụng của Knuth.

Quy ước: $i\le p\le q<j-1$

$$d_k(i,j) = d(i,k) + d(k+1,j) + c(i,j)$$

01.13.07.1 Chứng minh $opt(i,j-1) \le opt(i,j)$

Khi d thỏa mãn điều kiện Knuth và hàm chi phí thỏa mãn QI thì d cũng thỏa mãn QI.

Sử dụng QI cho các chỉ số: $p+1 \le q+1 \le j-1 < j$

$\Rightarrow d(p+1,j-1) + d(q+1,j) \le d(q+1,j-1) + d(p+1,j)$

$\Rightarrow d(i,p) + dp(p+1,j-1) + C(i,j-1) + d(i,q) + d(q+1,j) + C(i,j)$
$\le d(i,q) + d(q+1,j-1) + C(i,j-1) + d(i,p) + d(p+1,j) + C(i,j)$

$\Rightarrow d_p(i,j-1) + d_q(i,j) \le d_p(i,j) + d_q(i,j-1)$

$\Rightarrow d_p(i,j-1) - d_q(i,j-1) \le d_p(i,j) - d_q(i,j)$

Nếu $d_p(i,j-1) \ge d_q(i,j-1)$

Suy ra: $0 \le d_p(i,j-1) - d_q(i,j-1) \le d_p(i,j) - d_q(i,j) \Rightarrow d_p(i,j) \ge d_q(i,j)$

Vậy: nếu $d_p(i,j-1) \ge d_q(i,j-1)$ thì $d_p(i,j) \ge d_q(i,j)$ (1)

Ta xét $q = opt(i,j-1) \Rightarrow d_p(i,j-1) \ge d_q(i,j-1)\ \forall i \le p \le q$ (quy ước)

Suy ra: $d_p(i,j) \ge d_q(i,j)\ \forall i \le p \le q$ (chứng minh ở 1)

Vậy: $opt(i,j)$ nhỏ nhất là bằng q, hay $opt(i,j-1)$

$\Rightarrow opt(i,j-1) \le opt(i,j)$ (đccm)

01.13.07.2 Chứng minh $opt(i,j) \le opt(i+1,j)$

Tương tự như chứng minh $opt(i,j-1) \le opt(i,j)$

Sử dụng QI cho các chỉ số: $i \le i+1 \le p \le q$

$\Rightarrow d(i,\ p) + d(i+1,q) \le d(i,q) + d(i+1,p)$

$\Rightarrow d(i,p) + d(p+1,j) + C(i,j) + d(i+1,q) + d(q+1,j) + C(i+1,j)$
$\le d(i+1,p) + d(p+1,j) + C(i+1,j) + d(i,q) + d(q+1,j) + C(i,j)$

$\Rightarrow d_p(i,j) + d_q(i+1,j) \le d_p(i+1,j) + d_q(i,j)$

$\Rightarrow d_p(i,j) - d_q(i,\ j) \le d_p(i+1,j) - d_q(i+1,j)$

Nếu $d_p(i,j) \ge d_q(i,j)$

Suy ra: $0 \le d_p(i,j) - d_q(i,j) \le d_p(i+1,j) - d_q(i+1,j)$

$\Rightarrow d_p(i+1,j) \ge d_q(i+1,j)$

Vậy: nếu $d_p(i,j) \ge d_q(i,j)$ thì $d_p(i+1,j) \ge d_q(i+1,j)$ (1)

Ta xét $q = opt(i,j) \Rightarrow d_p(i,j) \ge d_q(i,j)\ \forall i \le p \le q$ (quy ước)

Suy ra: $d_p(i+1,j) \ge d_q(i+1,j)\ \forall i \le p \le q$ (chứng minh ở 1)

Vậy: $opt(i+1,j)$ nhỏ nhất là q, hay $opt(i,j)$

$\Rightarrow opt(i,j) \le opt(i+1,j)$ (đccm)

TÀI LIỆU THAM KHẢO

Nội dung	Nguồn tham khảo
Fibonacci	Dynamic programming - Wikipedia
Prefix Calculation	Prefix Sum Array - Implementation and Applications in Competitive Programming - GeeksforGeeks
	Introduction to Prefix Sums · USACO Guide
	Prefix sum - Wikipedia
	Prefix Sum Array Explained - YouTube
	Mảng cộng dồn và mảng hiệu (vnoi.info)
Longest Increasing Subsequences	Longest Increasing Subsequence | DP-3 - GeeksforGeeks
	Longest Increasing Subsequence - Interview Problem (afteracademy.com)
Dynamic Programming on Grid	Dynamic Programming - Problems involving Grids | HackerEarth
	Dynamic Programming : Grid Paths - YouTube
Knapsack	Knapsack problem - Wikipedia
	0-1 Knapsack Problem (Dynamic Programming) - YouTube
Longest Common Subsequence	Longest Common Subsequence (programiz.com)
	Longest Common Subsequence | DP-4 - GeeksforGeeks
	Longest increasing subsequence - Algorithms for Competitive Programming (cp-algorithms.com)
	Longest common subsequence problem - Wikipedia
	Longest Common Subsequence (tutorialspoint.com)
	Longest Common Subsequence Problem (techiedelight.com)
	4.9 Longest Common Subsequence (LCS) - Recursion and Dynamic Programming - YouTube
Dynamic Programming on Tree	Dynamic Programming on Trees | Set-1 - GeeksforGeeks
	Dynamic Programming on Trees | Set 2 - GeeksforGeeks
Sparse Table	Sparse Table - GeeksforGeeks
	Sparse Table - Algorithms for Competitive Programming (cp-algorithms.com)
Matrix Exponentiation	Binary Exponentiation - Algorithms for Competitive Programming (cp-algorithms.com)
	Matrix Exponentiation - GeeksforGeeks
	Matrix exponentiation | HackerEarth

	Matrix exponential - Wikipedia
	Fibonacci Numbers in O(logn) [Matrix Exponentiation] - Only Code
	Solving the Fibonacci Sequence with Matrix Exponentiation - YouTube
	What is Fast Exponentiation? - YouTube
	Matrix Exponentiation · USACO Guide
Dynamic Programming with Bitmask	Bitmasking and Dynamic Programming | Set 1 (Count ways to assign unique cap to every person) - GeeksforGeeks
	Bitmasking and Dynamic Programming | Set-2 (TSP) - GeeksforGeeks
Knuth's Optimization	Knuth's Optimization in Dynamic Programming - GeeksforGeeks
	Knuth's Optimization - Algorithms for Competitive Programming (cp-algorithms.com)
	Efficient dynamic programming using quadrangle inequalities | Proceedings of the twelfth annual ACM symposium on Theory of computing
	10003.pdf (onlinejudge.org) (Cutting Sticks)
Khác	Dynamic programming - Wikipedia
	5 Simple Steps for Solving Dynamic Programming Problems - YouTube
	Dynamic Programming lectures - YouTube

BẢNG PHÂN CÔNG CÔNG VIỆC

Sinh viên	MSSV	Công việc	Mức độ đóng góp
Nguyễn Đức Anh Phúc	20520276	Fibonacci Dynamic Programming on Grid Dynamic Programming on Tree Dynamic Programming with Bitmask	100%
Ngô Văn Tấn Lưu	20521591	Longest Increasing Subsequences Sparse Table	100%
Trương Thành Thắng	20521907	Knapsack Knuth's Optimization	100%
Huỳnh Viết Tuấn Kiệt	20521907	Prefix Calculation Longest Common Subsequence Matrix Exponentiation	100%