**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 4](#_Toc108783516)

[01.01 Giới thiệu về quy hoạch động 4](#_Toc108783517)

[01.01.01 Phương pháp quy hoạch động 4](#_Toc108783518)

[01.01.02 Dấu hiệu sử dụng Quy hoạch động 4](#_Toc108783519)

[01.01.03 Ý tưởng quy hoạch động 4](#_Toc108783520)

[01.01.04 Các bước thực hiện 4](#_Toc108783521)

[01.01.05 Nhận xét 4](#_Toc108783522)

[01.02 Các bài toán ứng dụng 5](#_Toc108783523)

[MỘT SỐ BÀI TOÁN 6](#_Toc108783524)

[01.03 Bài toán 1: Fibonacci 6](#_Toc108783525)

[01.03.01 Mô tả bài toán 6](#_Toc108783526)

[01.03.02 Mô hình hóa bài toán 6](#_Toc108783527)

[01.03.03 Thiết kế thuật toán 6](#_Toc108783528)

[01.03.04 Ví dụ minh họa 7](#_Toc108783529)

[01.03.05 Phân tích độ phức tạp 8](#_Toc108783530)

[01.04 Bài toán 2: Prefix Calculation 9](#_Toc108783531)

[01.04.01 Đặt vấn đề 9](#_Toc108783532)

[01.04.02 Mô tả bài toán 9](#_Toc108783533)

[01.04.03 Mô hình hóa bài toán 10](#_Toc108783534)

[01.04.04 Thiết kế thuật toán 10](#_Toc108783535)

[01.04.05 Ví dụ minh họa 11](#_Toc108783536)

[01.04.06 Phân tích độ phức tạp 12](#_Toc108783537)

[01.05 Bài toán 3: Dynamic Programming on Grid 13](#_Toc108783538)

[01.05.01 Mô tả bài toán 13](#_Toc108783539)

[01.05.02 Mô hình hóa bài toán 13](#_Toc108783540)

[01.05.03 Thiết kế thuật toán 13](#_Toc108783541)

[01.05.04 Ví dụ minh họa 15](#_Toc108783542)

[01.05.05 Phân tích độ phức tạp 16](#_Toc108783543)

[01.06 Bài toán 4: Longest Increasing Subsequences 16](#_Toc108783544)

[01.06.01 Mô tả bài toán 16](#_Toc108783545)

[01.06.02 Mô hình hóa bài toán 16](#_Toc108783546)

[01.06.03 Thiết kế thuật toán 17](#_Toc108783547)

[01.06.04 Ví dụ minh họa 17](#_Toc108783548)

[01.06.05 Phân tích độ phức tạp 18](#_Toc108783549)

[01.07 Bài toán 5: Knapsack 19](#_Toc108783550)

[01.07.01 Mô tả bài toán 19](#_Toc108783551)

[01.07.02 Mô hình hóa bài toán 19](#_Toc108783552)

[01.07.03 Thiết kế thuật toán 19](#_Toc108783553)

[01.07.04 Ví dụ minh họa 21](#_Toc108783554)

[01.07.05 Phân tích độ phức tạp 22](#_Toc108783555)

[01.08 Bài toán 6: Longest Common Subsequence 22](#_Toc108783556)

[01.08.01 Mô tả bài toán 22](#_Toc108783557)

[01.08.02 Mô hình hóa bài toán 22](#_Toc108783558)

[01.08.03 Thiết kế thuật toán 22](#_Toc108783559)

[01.08.04 Ví dụ minh họa 26](#_Toc108783560)

[01.08.05 Truy xuất giá trị 26](#_Toc108783561)

[01.08.06 Phân tích độ phức tạp 27](#_Toc108783562)

[01.09 Bài toán 7: Dynamic Programming on Tree 27](#_Toc108783563)

[01.09.01 Mô tả bài toán 27](#_Toc108783564)

[01.09.02 Mô hình hóa bài toán 27](#_Toc108783565)

[01.09.03 Thiết kế thuật toán 28](#_Toc108783566)

[01.09.04 Ví dụ minh họa 30](#_Toc108783567)

[01.09.05 Phân tích độ phức tạp 30](#_Toc108783568)

[01.10 Bài toán 8: Sparse Table 31](#_Toc108783569)

[01.10.01 Mô tả bài toán 31](#_Toc108783570)

[01.10.02 Mô hình hóa bài toán 31](#_Toc108783571)

[01.10.03 Thiết kế thuật toán 32](#_Toc108783572)

[01.10.04 Ví dụ minh họa 34](#_Toc108783573)

[01.11 Bài toán 9: Matrix Exponentiation 36](#_Toc108783574)

[01.11.01 Mô tả bài toán 36](#_Toc108783575)

[01.11.02 Mô hình hóa bài toán 36](#_Toc108783576)

[01.11.03 Thiết kế thuật toán 36](#_Toc108783577)

[01.11.04 Ví dụ minh họa 38](#_Toc108783578)

[01.11.05 Phân tích độ phức tạp 39](#_Toc108783579)

[01.11.06 Bài toán ứng dụng 39](#_Toc108783580)

[01.12 Bài toán 10: Dynamic Programming with Bitmask 40](#_Toc108783581)

[01.12.01 Mô tả bài toán 40](#_Toc108783582)

[01.12.02 Mô hình hóa bài toán 40](#_Toc108783583)

[01.12.03 Thiết kế thuật toán 41](#_Toc108783584)

[01.12.04 Ví dụ minh họa 43](#_Toc108783585)

[01.12.05 Phân tích độ phức tạp 44](#_Toc108783586)

[01.13 Bài toán 11: Knuth’s optimization 44](#_Toc108783587)

[01.13.01 Knuth’s optimization 44](#_Toc108783588)

[01.13.02 Mô tả bài toán ứng dụng (Cutting Sticks) 45](#_Toc108783589)

[01.13.03 Mô hình hóa bài toán 45](#_Toc108783590)

[01.13.04 Thiết kế thuật toán 45](#_Toc108783591)

[01.13.05 Ví dụ minh họa 47](#_Toc108783592)

[01.13.06 Phân tích độ phức tạp 48](#_Toc108783593)

[01.13.07 Chứng minh tính đúng đắn của Knuth’s optimization 48](#_Toc108783594)

[Tài liệu tham khảo 50](#_Toc108783595)

[Bảng phân công công việc 52](#_Toc108783596)

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

## Giới thiệu về quy hoạch động

### 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.

### 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 để 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.

### 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.

### 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.

### Nhận xét

#### 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.

#### 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.

## 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

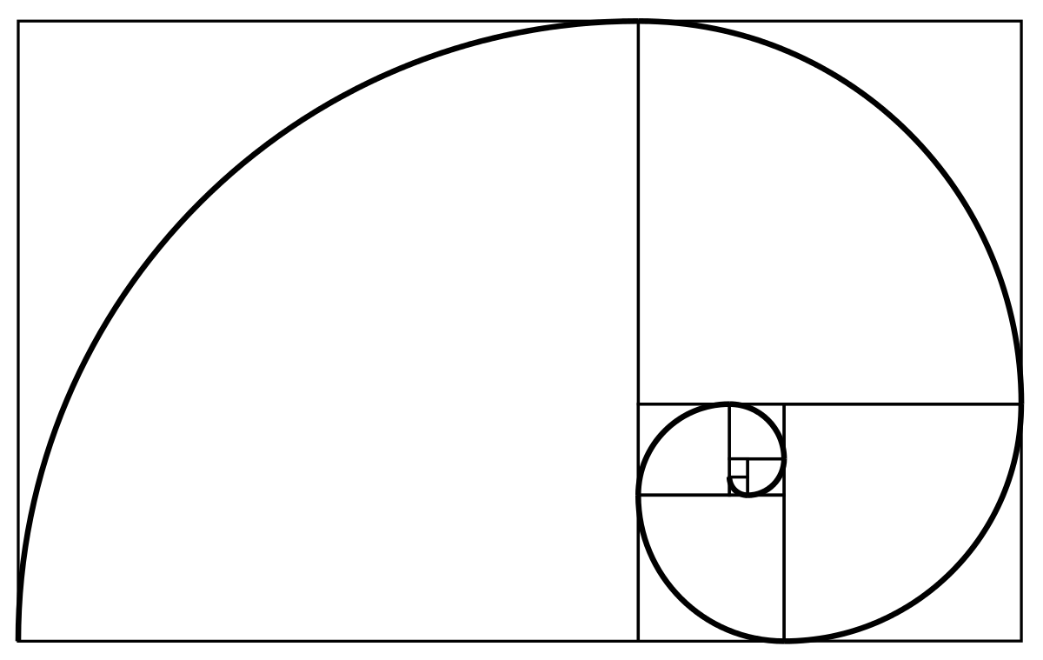
# MỘT SỐ BÀI TOÁN

## Bài toán 1: Fibonacci

### Mô tả bài toán

Cho số nguyên dương . Tìm số Fibonacci thứ .

số fibnacci đầu tiên là :



### Mô hình hóa bài toán

#### Input:

* dòng duy nhất chứa một số nguyên dương là số Fibonacci thứ

#### Output:

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

#### Ví dụ:

|  |  |
| --- | --- |
| **Input** | **Output** |
| 7 | 13 |

### Thiết kế thuật toán

* Công thức truy hồi số Fibonacci:
* Gọi là số Fibonacci thứ 𝑛:
* 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:
* Lý do: Overlapping Subproblem (giải nhiều lần một bài toán con) → Nhược điểm của chia để trị

Diagram

Description automatically generated

* 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:

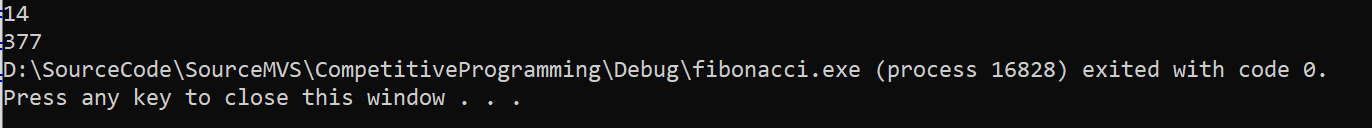
|  |
| --- |
| fib[]  fibo (n):  if fib[i] != undefined:  return fib[n]  if n==1 or n==2:  fib[n] = 1  else  tmp1 = fibo(n-1)  tmp2 = fibo(n-2)  fib[n] = tmp1 + tmp2  return fib[n] |

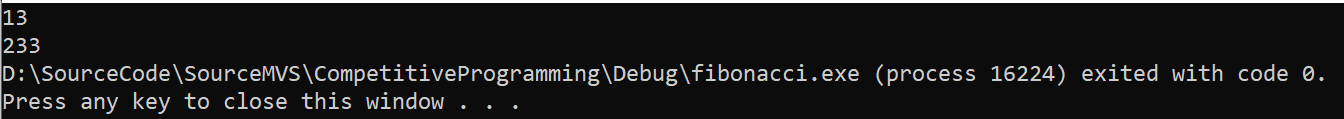
* Ngoài ra, còn có cách tiếp cận bằng phương pháp bottom-up

|  |
| --- |
| 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] |

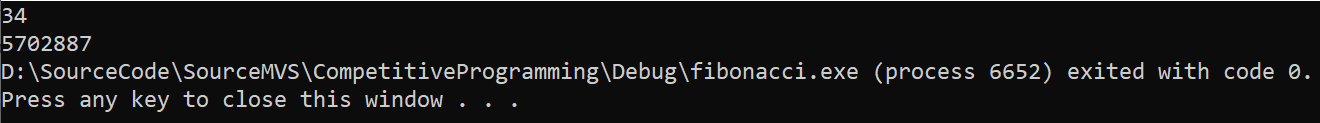
### Ví dụ minh họa

|  |  |
| --- | --- |
| **Input** | **Output** |
| 14 | 377 |





|  |  |
| --- | --- |
| **Input** | **Output** |
| 13 | 233 |



|  |  |
| --- | --- |
| **Input** | **Output** |
| 34 | 5702887 |

### 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 cho thời gian và không gian bộ nhớ là

## Bài toán 2: Prefix Calculation

### Đặt vấn đề

#### Bài toán 1 truy vấn

* Mô tả bài toán:
* Cho dãy gồm phần tử được đánh thứ tự từ . Cho câu truy vấn , tìm tổng các phần tử dãy trong đoạn .
* Mô hình hóa bài toán:
* Input:
* Cho số nguyên
* Cho dãy chứa các phần tử với
* Cho hai số nguyên: và
* Output:
* Kết quả
* 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 từ vị trí đến vị trí , cộng dồn giá trị tại mỗi vị trí và lưu vào biến kết quả.
* Độ phức tạp:

#### Đặt vấn đề

* Cũng với mô tả bài toán như trên, nhưng thay 1 truy vấn thành 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 và thuật toán này không hiệu quả trong trường hợp và đồng thời cực lớn (với là kích thước dãy và là số lượng truy vấn). Do đó, cần có một cách tiếp cận khác khả thi hơn.

### Mô tả bài toán

* Cho dãy gồm phần tử được đánh thứ tự từ . Cho câu truy vấn , với mỗi truy vấn, tìm tổng các phần tử dãy trong đoạn .

### Mô hình hóa bài toán

#### Input:

* Số nguyên dương : Số lượng phần tử của dãy
* Số nguyên dương : Số lượng câu truy vấn
* phần tử dãy , là giá trị tại vị trí trong dãy ()
* dòng, mỗi dòng hai số nguyên và (), vị trí thứ và trong dãy

#### Output

* dòng, mỗi dòng tương ứng với kết quả 1 truy vấn

### Thiết kế thuật toán

#### Ý tưởng

* Từ dãy đầu vào có phần tử
* Sử dụng mảng cộng dồn với
* Tổng

#### Phân tích đặc trưng Optimal Substructure, Overlapping Subproblem

* Gọi là tổng các phần tử dãy từ vị trí đến
* Cho 2 truy vấn:
* Việc sử dụng vòng lặp, lặp qua phép tính trongtruy vấn là không cần thiết, vì phép tính đã được thực thi ở truy vấn . Truy vấn có thể được biểu diễn lại như sau:

Đây là đặc trưng Overlapping Subproblems

* Dãy có

Đây là đặc trưng Optimal Substructure

#### Xác định phương trình quy hoạch động

* Gọi là tổng các phần tử của dãy từ vị trí đến

Và

* Bài toán cơ sở

theo định nghĩa, là tổng các phần tử của dãy từ đến , nên , mà từ , như vậy thỏa điều kiện (2)

* Từ và , xây dựng được phương trình quy hoạch động

#### Tạo bảng và lưu giá trị

* Từ dãy đầu vào có phần tử
* Tạo bảng lưu trữ, gọi là mảng (viết tắt của ) kích thước bằng mảng , giá trị bằng tổng các giá trị phần tử của dãy từ vị trí đến vị trí

|  |
| --- |
| 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 |

#### Tra bảng và xây dựng lời giải

* Từ mỗi truy vấn với hai giá trị , ta có kết quả mỗi truy vấn

trong đó là dãy đã khởi tạo ở mục **02.01.04.3**

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

### 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 |
| Text  Description automatically generated | Text  Description automatically generated | Text  Description automatically generated |
| Kết quả testcase 1 | Kết quả testcase 2 | Kết quả testcase 3 |

### Phân tích độ phức tạp

* Tạo mảng cần duyệt qua phần tử mảng :
* Tiếp theo duyệt qua truy vấn:
* Mỗi truy vấn chỉ thực hiện tính toán thông qua truy xuất giá trị từ mảng :
* Độ phức tạp thuật toán:

## Bài toán 3: Dynamic Programming on Grid

### 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 ô ( hàng và 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 xe có thể di chuyển sang một trong số các ô nếu ô đó tồn tại. Thời gian vượt qua ô là . .

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

A picture containing text, clock, clipart

Description automatically generated

Table

Description automatically generated

### Mô hình hóa bài toán

* Input:
* Dòng đầu tiên chứa 2 số nguyên và
* Dòng thứ trong dòng sau chứa số nguyên
* 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 |

### 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

Diagram

Description automatically generated

* Nguyên lí tối ưu Bellman

Graphical user interface, text, application

Description automatically generated

* Điều khiển ở bước tiếp theo là tối ưu của điều khiển bước phía trước
* **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 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:
* Bài toán cơ sở:

Diagram, table

Description automatically generated

* 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 |

### Text Description automatically generatedVí dụ minh họa

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

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

Text

Description automatically generated

### 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
* Độ phức tạp không gian

## Bài toán 4: Longest Increasing Subsequences

### Mô tả bài toán

Cho dãy các số nguyên gồm phần tử, tìm độ dài của dãy con có độ dài lớn nhất của được sắp xếp theo thứ tự tăng dần. Cho biết một dãy con của là một cách chọn ra trong một số phần tử và giữ nguyên thứ tự.

### Mô hình hóa bài toán

#### Input:

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

#### 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 .

#### 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: . Dãy này có phần tử.

### Thiết kế thuật toán

* Gọi là độ dài của dãy con tăng dài nhất kết thúc tại (). Vì dãy con tăng dài nhất chưa biết kết thúc ở đâu trên đoạn từ đến nên ta sẽ tính toàn bộ () 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 , cần tìm dãy con tăng dài nhất kết thúc tại , đây là tính chất **Overlapping Subproblem.**
  + Để tìm được dãy con tăng dài nhất kết thúc tại cần tìm được dãy con tăng dài nhất kết thúc ở đâu đó trước . Đây là tính chất **Optimal Substructure.**
* Bài toán cơ sở: . Vì tại vị trí có , dãy con này chỉ có một phần tử.
* Dãy con tăng dài nhất kết thúc ở vị trí sẽ được thành lập bằng cách lấy ghép vào đuôi của một trong số những dãy con tăng dài nhất bắt đầu tại vị trí đứng trước . Vì chỉ được ghép vào đuôi những dãy con bắt đầu tại nào đó nhỏ hơn để đảm bảo được tính tăng và ta sẽ phải chọn dãy dài nhất để ghép vào đầu. Như vậy, có thể được tính như sau:
  + Xét tất cả các chỉ số trong khoảng từ đến mà , chọn ra chỉ số có lớn nhất gọi là , sau đó tính . 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); |

### Ví dụ minh họa

* Test case 1:

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

Text

Description automatically generated

* Text

  Description automatically generatedTest case 2:

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

* Text

  Description automatically generatedTest case 3:

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

### Phân tích độ phức tạp

Để xây dựng được dãy , ta cần vòng lặp thứ nhất lặp qua từng phần tử của dãy . Ở mỗi phần tử của dãy , để tìm được ta cần lặp qua từ đến . Mà độ dài của mảng là . Do vậy, độ phức tạp của ý tưởng này là .

## Bài toán 5: Knapsack

### Mô tả bài toán

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

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

### Mô hình hóa bài toán

#### Input:

* 1 số nguyên là số lượng đồ vật .
* 1 mảng số nguyên có phần tử, là trọng lượng của đồ vật thứ . .
* 1 mảng số nguyên có phần tử, là giá trị của đồ vật thứ
* 1 số nguyên là trọng lượng tối đa mà túi có thể đựng

#### Output:

* 1 số nguyên là tổng giá tri tối đa mà có thể lấy được.
* 1 mảng nhị phân có phần tử ( là chọn đồ vật thứ , và ngược lại)
* Hàm mục tiêu:

#### Constraints:

### Thiết kế thuật toán

#### Phân tích đặc trưng optimal substructure, overlapping subproblems

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

Mối quan hệ giữa các trạng thái:

Khi đó, ta muốn d(i,m) đạt giá trị tối ưu, thì dĩ nhiên , 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:

với

Ta thấy 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.

#### Xác định phương trình quy hoạch động

Khi , không có vật vào để chọn cả giá trị túi là .

Khi , túi khổng thể chứa vật nào cả giá trị túi là .

Phương trình Quy hoạch động:

#### Tạo bảng và lưu giá trị

|  |
| --- |
| init matrix has rows, 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 |

#### Tra bảng, xây dựng lời giải

Tổng giá trị tối đa có thể lấy được trong bài toán chính là

|  |
| --- |
| print d[n, M] |

Phương án chọn đồ vật:

|  |
| --- |
| init array sol has 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 |

### Ví dụ minh họa

|  |  |
| --- | --- |
| **Input** | **Output** |
| 5  12 2 1 4 1  4 2 1 10 2  15 | 15  0 1 1 1 1 |

Text

Description automatically generated

|  |  |
| --- | --- |
| **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 |

Text

Description automatically generated

|  |  |
| --- | --- |
| **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 |

Text

Description automatically generated

### Phân tích độ phức tạp

Bước tạo bảng:

Bước xây dựng lời giải:

Độ phức tạp của giải thuật:

## Bài toán 6: Longest Common Subsequence

### Mô tả bài toán

* Cho 2 dãy và , 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.

### Mô hình hóa bài toán

#### Input

* 2 chuỗi

#### Output

* Số nguyên : Độ dài chuỗi con chung dài nhất của hai dãy

### Thiết kế thuật toán

#### Ý tưởng giải quyết

* Gọi là chuỗi được tạo từ các phần tử của chuỗi từ vị trí đến vị trí
* Gọi là độ dài chuỗi con chung dài nhất của hai chuỗi
* Đầu vào bài toán:
* Chuỗi kích thước
* Chuỗi kích thước
* Ý tưởng giải quyết: So kí tự cuối cùng trong hai dãy
* Trường hợp 1: Kí tự cuối cùng của 2 dãy bằng nhau
* Trường hợp 2: Kí tự cuối cùng của 2 dãy khác nhau

#### Phân tích đặc trưng Optimal Substructure, Overlapping Subproblem

* Xét 2 công thức đưa ra ở mục **02.02.03.1** với 2 chuỗi và
* 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 và đạt giá trị lớn nhất thì độ dài dãy con chung của 2 dãy và 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 ra khỏi chuỗi , tìm độ dài dãy con chung dài nhất của chuỗi với chuỗi . Giai đoạn 2, loại bỏ kí tự cuối cùng ra khỏi chuỗi , tìm độ dài dãy con chung dài nhất của chuỗi với chuỗi . Khi đó độ dài dãy con chung của 2 dãy và đạt giá trị lớn nhất thì độ dài dãy con chung khi loại bỏ khỏi chuỗi hoặc khỏi chuỗi phải lớn nhất.
* Từ phân tích trên Đây là đặc trưng Optimal Substructure
* Dựa vào 2 công thức đưa ra ở **02.02.03.1**, 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 là kích thước , là kích thước
* hoặc , một trong hai dãy rỗng dừng thuật toán
* Minh họa ví dụ
* Cho dãy
* Cho dãy
* Tìm
* Xây dựng cây đệ quy

Chart

Description automatically generated

* Trong đó, chấm đen thể hiện độ dài dãy con chung dài nhất của 2 dãy đang xét
* Dựa trên hình, (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 Đây là đặc trưng Overlapping Subproblem
* Có thể giải bài toán bằng quy hoạch động

#### Xác định phương trình quy hoạch động

* Trong đó
* là độ dài dãy con chung dài nhất của 2 dãy và
* là dãy được tạo từ dãy bằng cách giữ lại các phần tử từ trong dãy
* là phần tử thứ của dãy
* Trường hợp khi hoặc là bài toán cơ sở của vấn đề

#### Tạo bảng và lưu giá trị

* Cho 2 chuỗi đầu vào có kích thước lần lượt
* Khởi tạo bảng lưu trữ giá trị có kích thước hoặc , bảng lưu trữ là một mảng hai chiều
* Dựa vào phương trình quy hoạch động ở **02.02.03.3**, xây dựng và lưu trữ kết quả như sau:
* Nếu
* Nếu

|  |
| --- |
| 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 |

#### Tra bảng và xây dựng lời giải

* Xét bảng được tạo ở **02.02.03.4**, mỗi ô trong bảng thể hiện với là dãy được tạo từ dãy bằng cách giữ lại các phần tử từ trong dãy .
* Bài toán cần tính , tức là với là kích thước dãy và là kích thước . Theo bảng lưu trữ, là kết quả được lưu trữ ở ô , do đó, ngay sau giai đoạn tạo bảng lưu trữ có thể trả về ngay kết quả ô 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] |

### Ví dụ minh họa

|  |  |  |
| --- | --- | --- |
|  | | |
| |  | | --- | | Input | | codeforces  conference |   Testcase 1 | |  | | --- | | Input | | aaaaaaaaaaa  abababab |   Testcase 2 | |  | | --- | | Input | | lcslcslcslcs  sclsclsclscl |   Testcase 3 |
| Text  Description automatically generated | Text  Description automatically generated | Text  Description automatically generated |
| Kết quả testcase 1 | Kết quả testcase 2 | Kết quả testcase 3 |

### 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 là dòng cuối cùng của bảng, là cột cuối cùng của bảng, là ô nằm góc phải dưới của bảng (giá trị ô là kết quả của bài toán)
* Khởi tạo stack rỗng
* Khởi tạo vòng lặp với điều kiện lặp và
* Nếu (1): Nhảy lên ô trên
* Nếu (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 : Thêm phần tử (hoặc ) 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 và .

|  |
| --- |
| 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 |

### Phân tích độ phức tạp

* 2 chuỗi đầu vào có kích thước lần lượt
* Bảng lưu trữ được biểu diễn dưới dạng mảng 2 chiều có kích thước
* Duyệt toàn bộ mảng hai chiều để tính giá trị mỗi ô tương ứng
* Độ phức tạp thuật toán:

## 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

### Mô tả bài toán

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

Diagram

Description automatically generated

* Kết quả là những subtree

### Mô hình hóa bài toán

#### Input:

* Dòng đầu tiên chứa số nguyên dương , số lượng các node
* dòng tiếp theo chứa 2 số nguyên dương là cạnh nối đỉnh và
* Dòng tiếp theo chứa số nguyên là màu của các node

#### Output:

* Các subtree số lượng màu bằng nhau

### 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 tính cả node gốc
* Nếu số lượng node 0 bằng số lượng node 1 thì là valid subtree root 𝑢
* Độ phức tạp:
* 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 |

Icon

Description automatically generated with medium confidence

* 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:
* là các node con của node 𝑢
* Kết quả của node 𝑢 sẽ là:

(𝑣 𝑙à 𝑐𝑜𝑛 𝑡𝑟ự𝑐 𝑡𝑖ế𝑝 𝑐ủ𝑎 𝑢)

Shape

Description automatically generated

* 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
* Gọi là số lượng node có màu là 𝑗 trong subtree gốc 𝑖
* Công thức truy hồi:

Diagram

Description automatically generated

* Bài toán cơ sở:

Text

Description automatically generated

* 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 thì phải biết kết quả của các node con
* 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 |

### 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

Table, calendar

Description automatically generated

Text

Description automatically generated

### Phân tích độ phức tạp

* Độ phức tạp của thuật toán sẽ là 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

## 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 , 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 .

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.

### Mô tả bài toán

Cho dãy các số nguyên gồm phần tử. Cho khoảng truy vấn gồm hai số nguyên và với . Với mỗi khoảng truy vấn, hãy cho biết giá trị nhỏ nhất của các phần tử trong mảng trong đoạn .

### Mô hình hóa bài toán

#### Input:

* Dòng thứ nhất chứa một số nguyên dương là độ dài của dãy đầu vào.
* Dòng thứ hai chứa số, là các phần tử trong dãy .
* Dòng thứ ba chứa một số nguyên dương là số khoảng truy vấn.
* dòng tiếp theo, mỗi dòng chứa hai số nguyên dương và ()

#### Output:

* dòng, mỗi dòng là giá trị nhỏ nhất trong khoảng của mảng .
* Lưu ý: số thứ tự trong mảng bắt đầu từ .

#### Ví dụ:

|  |  |
| --- | --- |
| **Input** | **Output** |
| 3  1 4 1  2  1 1  1 2 | 4  1 |

* Giải thích:
  + Ở khoảng truy vấn đầu tiên, mảng trong khoảng chỉ có một phần tử là nên giá trị nhỏ nhất của đoạn là .
  + Ở khoảng truy vấn thứ hai, mảng trong khoảng có hai phần tử là và giá trị nhỏ nhất của đoạn là .

### 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 . Bảng không nhất thiết phải được lấp đầy chính vì vậy nên được gọi là Sparse Table.

Với mỗi khoảng truy vấn 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 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 . 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 . Với mỗi số nguyên dương bất kỳ, ta có thể phân tích thành tổng các số nguyên là lũy thừa của . Ví dụ:

Ta lại có: Với mỗi đoạn bất kỳ, độ dài của đoạn là . Do đó, ta có thể chia đoạn thành các đoạn nhỏ có độ dài là lũy thừa của .Ví dụ:

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

1. Tìm được giá trị nhỏ nhất của các đoạn trong khoảng sẽ tìm được giá trị nhỏ nhất trong khoảng **Optimal substructure**
2. Ở mỗi đoạn trong khoảng 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 , 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.

#### 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 ), chỉ vị trí bắt đầu của khoảng trong mảng. Chỉ số hàng là lũy thừa là độ dài của một đoạn. Ví dụ: hàng sẽ lưu giá trị nhỏ nhất của các đoạn có độ dài là , cột sẽ lưu giá trị nhỏ nhất của các đoạn có vị trí bắt đầu là 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 , ta có thể sử dụng đệ quy để chia đoạn thành nhiều khoảng để lấy ra giá trị nhỏ nhất của từng khoảng trong bảng với độ phức tạp . Tuy nhiên, ta có thể chia đoạn 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 sau:

Có thể thấy đoạn và đoạn có phần tử trùng là phần tử tại , nhưng phần trùng này sẽ không ảnh hương kết quả do vậy ta có thể tận dụng để thực hiện truy vấn với độ phức tạp vì hai đoạn này có độ dài bằng nhau (bằng ) 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 .

Tổng quát, với mỗi đoạn , ta có thể thực hiện so sánh giữa hai ô trong bảng Sparse Table là ô tại vị trí hàng cột với ô tại vị trí hàng cột .

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]); |

### Ví dụ minh họa

* Test case 1:

|  |  |
| --- | --- |
| **Input** | **Output** |
| 3  1 4 1  2  1 1  1 2 | 4  1 |

Text

Description automatically generated

* Text

  Description automatically generatedTest case 2:

|  |  |
| --- | --- |
| **Input** | **Output** |
| 5  1 5 2 4 3  5  1 5  1 3  3 5  3 6  1 5 | 1  1  2  0  1 |

* Text

  Description automatically generatedTest 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 |

## Bài toán 9: Matrix Exponentiation

### Mô tả bài toán

* Cho biểu thức , với có thể là một con số, một đa thức, một ma trận, …, tính lũy thừa bậc của với 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.

### Mô hình hóa bài toán

#### Input

* Số nguyên dương : Kích thước ma trận
* Số nguyên dương : Số mũ của phép lũy thừa
* Ma trận : Cơ số của phép lũy thừa

#### Output

* Kết quả phép tính

#### Constraint

* Kích thước ma trận Ma trận vuông

### Thiết kế thuật toán

#### Phân tích đặc trưng Optimal Substructure, Overlapping Subproblem

* Cho ma trận đầu vào và số nguyên là số mũ phép lũy thừa:

thừa số

* Độ phức tạp khi nhân thừa số với nhau: (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:
* được tính bằng cách gom thành nhiều nhóm, các nhóm được biễu diễn với
* Công thức được biểu diễn theo mô hình dưới đây:

Diagram

Description automatically generated

* Theo mô hình trên, mỗi nhóm phải tính tích thừa số. Nếu có
* Biểu diễn mô hình trên yêu cầu tính lặp lại các bài toán nhiều lần Đây là đặc trưng Overlapping Subproblem.
* Mặc khác
* Nếu đã có ,
* Nếu đã có ,
* …
* Với biểu diễn công thức trên, để tối thiểu hóa số phép tính , yêu cầu phải tối thiểu hóa số phép tính của Đây là đặc trưng Optimal Substructure.

#### Tạo bảng và lưu giá trị

* Cho đầu vào ma trận với số mũ
* Tạo mảng tích lũy chứa phần tử

|  |
| --- |
| 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 |

#### 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 |

### Ví dụ minh họa

|  |  |
| --- | --- |
|  | Text  Description automatically generated |
|  |  |
|  | Text  Description automatically generated |
|  |  |
|  | Text  Description automatically generated |

### Phân tích độ phức tạp

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

### Bài toán ứng dụng

* Bài toán tìm số Fibonacci đã có giải pháp áp dụng quy hoạch động với độ phức tạp . Tuy nhiên, với các trường hợp cực lớn (chẳng hạn ), giải pháp 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:
* Biểu diễn công thức Fibonacci dưới dạng phép nhân ma trận:
* Công thức sử dụng ma trận không vuông, do đó chưa thể áp dụng ma trận lũy thừa
* Biểu diễn khác:
* Ta có:


* …
* Vậy công thức Fibonacci đã biểu diễn được dưới dạng lũy thừa của ma trận . Như vậy đã có thể tính số Fibonacci với độ phức tạp
* Minh họa

|  |  |
| --- | --- |
| Text  Description automatically generated | Text  Description automatically generated |

## Bài toán 10: Dynamic Programming with Bitmask

### 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

A picture containing radar chart

Description automatically generated

* Mô hình hóa: Đồ thị vô hướng || có hướng
* Mục tiêu: Tìm chu trình Hamilton ngắn nhất

Chart, line chart

Description automatically generated

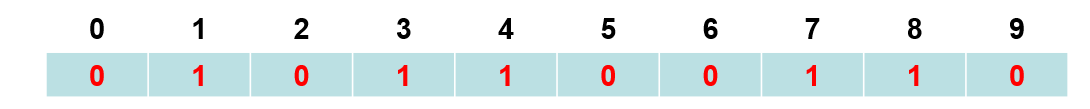
### Mô hình hóa bài toán

* Input:
* Dòng đầu tiên chứa 2 số nguyên dương và là số đỉnh và số cạnh của đồ thị
* dòng tiếp theo chứa 3 số nguyên là cạnh và trọng số của đồ thị ()
* 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 |

### 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:

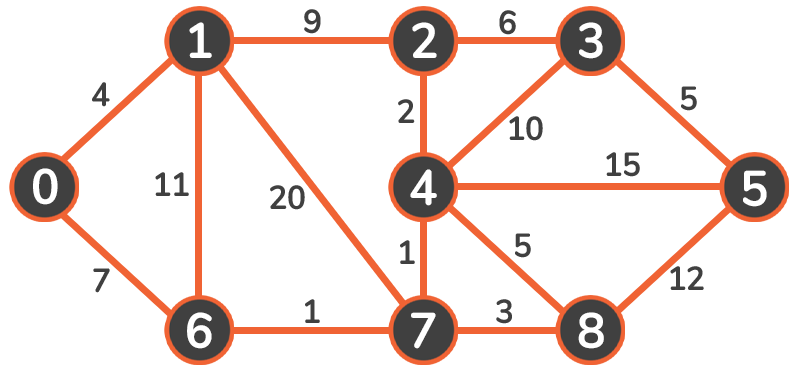


* Xét một đường đi bất kì, mỗi đỉnh trong đồ thị có thể thuộc đường đi hoặc không
* Với hai đường đi có cùng tập đỉnh (cùng cấu hình) và cùng đỉnh kết thực, khả năng mở rộng của chúng là như nhau. Trong các đường đi có cùng cấu hình và cùng điểm kết thúc, ta chỉ quan tâm các đường đi có chi phí nhỏ nhất: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

A picture containing text, outdoor, traffic, light

Description automatically generated

* 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:



* Có nhiều cấu hình giống nhau, chúng được phân biệt với nhau bằng đỉnh kết thực **Overlapping subproblem**
* Công thức truy hồi:
* Bài toán cơ sở: Do chu trình luôn bắt đầu từ 0 nên
* Ví dụ:
* Xét cấu hình kết thúc tại
* Cấu hình kết thúc tại
* Cấu hình kết thúc tại
* 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]) |

### 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

A picture containing text, clock

Description automatically generated

* Bảng phương án:

Table

Description automatically generated

Text

Description automatically generated with medium confidence

### Phân tích độ phức tạp

* Tổng số lượng cấu hình là : . Với mỗi cấu hình lặp cặp đỉnh .
* Độ phức tạp cảu thuật toán là

## Bài toán 11: Knuth’s optimization

### Knuth’s optimization

#### 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 thành .

#### Điều kiện áp dụng

* Giải pháp quy hoạch động có dạng:
* Hàm chi phí thỏa mãn điều kiện:

(The quadrangle inequality [QI])

với

#### Hệ quả:

Gọi là giá trị của mà tối thiểu hóa . Khi đó, Kỹ thuật tối ưu Knuth sẽ giới hạn miền tìm kiếm . Từ sang

### 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 phải được cắt ở cách vạch , , tính từ một phía. Có một số lựa chọn. Người ta có thể cắt đầu tiên vào vạch , sau đó và cuối cùng là . Điều này dẫn đến chi phí là , vì thanh đầu tiên là , sau đó là và cuối cùng là . Lựa chọn khác có thể là cắt ở vạch , sau đó là , cuối cùng là . Điều này dẫn đến chi phí là , đây là một phương án tốt hơn.

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

### Mô hình hóa bài toán

#### Input:

* 1 số nguyên dương là độ dài của gậy gỗ cần cắt.
* 1 số nguyên là số vạch cần cắt.
* 1 mảng số nguyên dương có phần tử, mỗi phần tử là vị trí trên gậy gỗ cần phải cắt, các phần tử được sắp xếp theo thứ tự tăng dần.

#### Output:

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

### Thiết kế thuật toán

#### 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.

#### Xác định phương trình quy hoạch động

Gọi là chi phí tối thiểu để cắt cây gậy có độ dài bắt đầu từ vạch thứ đến vạch thứ .

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 và . Có nghĩa là mảng giờ có phần tử và bài toán là tìm .

Phương trình quy hoạch động:

Trong đó là chi phí cắt cây gậy có độ dài bắt đầu từ vạch thứ đến vạch thứ .

#### Tạo bảng và lưu giá trị

|  |
| --- |
| insert 0 to head of array  insert to tail of array  init matrix with rows, 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 do:  d[i][j] = min(d[i][j], d[i][k]+d[k][j]+c[j]-c[i])  end  end  end |

#### Tra bảng, xây dựng lời giải ban đầu

|  |
| --- |
| print d[0][n+1] |

#### Áp dụng Knuth’s optimization

Kiểm tra điều kiện áp dụng:

* Xét giải pháp quy hoạch động:

thỏa mãn điều kiện

* Xét hàm chi phí:

Với 4 số (mảng tăng dần)

Ta có:

(1)

Mặt khác:

(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 đến , ta tìm trong đoạn từ đến .

|  |
| --- |
| insert 0 to head of array  insert to tail of array  init matrix with rows, columns  init matrix with rows, 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 |

### Ví dụ minh họa

|  |  |
| --- | --- |
| **Input** | **Output** |
| 100  3  25 50 75 | 200 |

Graphical user interface, text

Description automatically generated

|  |  |
| --- | --- |
| **Input** | **Output** |
| 10  4  4 5 7 8 | 22 |

Graphical user interface, text

Description automatically generated with medium confidence

|  |  |
| --- | --- |
| **Input** | **Output** |
| 20  3  5 10 15 | 40 |

Graphical user interface, text

Description automatically generated with medium confidence

### Phân tích độ phức tạp

Giải pháp trên có độ phức tạp:

### Chứng minh tính đúng đắn của Knuth’s optimization

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

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

Quy ước:

#### Chứng minh

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

Sử dụng QI cho các chỉ số:

Nếu

Suy ra:

Vậy: nếu thì (1)

Ta xét

Suy ra: (chứng minh ở 1)

Vậy: nhỏ nhất là bằng , hay

(đccm)

#### Chứng minh

Tương tự như chứng minh

Sử dụng QI cho các chỉ số:

Nếu

Suy ra:

Vậy: nếu thì (1)

Ta xét

Suy ra: (chứng minh ở 1)

Vậy: nhỏ nhất là , hay

(đccm)

# Tài liệu tham khảo

|  |  |
| --- | --- |
| **Nội dung** | **Nguồn tham khảo** |
| Fibonacci | [Dynamic programming - Wikipedia](https://en.wikipedia.org/wiki/Dynamic_programming) |
| Prefix Calculation | [Prefix Sum Array - Implementation and Applications in Competitive Programming - GeeksforGeeks](https://www.geeksforgeeks.org/prefix-sum-array-implementation-applications-competitive-programming/) |
| [Introduction to Prefix Sums · USACO Guide](https://usaco.guide/silver/prefix-sums?lang=cpp) |
| [Prefix sum - Wikipedia](https://en.wikipedia.org/wiki/Prefix_sum) |
| [Prefix Sum Array Explained - YouTube](https://www.youtube.com/watch?v=7pJo_rM0z_s&t=144s) |
| [Mảng cộng dồn và mảng hiệu (vnoi.info)](https://vnoi.info/wiki/algo/data-structures/prefix-sum-and-difference-array.md) |
| Longest Increasing Subsequences | [Longest Increasing Subsequence | DP-3 - GeeksforGeeks](https://www.geeksforgeeks.org/longest-increasing-subsequence-dp-3/) |
| [Longest Increasing Subsequence - Interview Problem (afteracademy.com)](https://afteracademy.com/blog/longest-increasing-subsequence) |
| Dynamic Programming on Grid | [Dynamic Programming - Problems involving Grids | HackerEarth](https://www.hackerearth.com/practice/notes/dynamic-programming-problems-involving-grids/) |
| [Dynamic Programming : Grid Paths - YouTube](https://www.youtube.com/watch?v=V64F4wlodUM) |
| Knapsack | [Knapsack problem - Wikipedia](https://en.wikipedia.org/wiki/Knapsack_problem) |
| [0-1 Knapsack Problem (Dynamic Programming) - YouTube](https://www.youtube.com/watch?v=xOlhR_2QCXY) |
| Longest Common Subsequence | [Longest Common Subsequence (programiz.com)](https://www.programiz.com/dsa/longest-common-subsequence#:~:text=The%20longest%20common%20subsequence%20(LCS,positions%20within%20the%20original%20sequences.) |
| [Longest Common Subsequence | DP-4 - GeeksforGeeks](https://www.geeksforgeeks.org/longest-common-subsequence-dp-4/) |
| [Longest increasing subsequence - Algorithms for Competitive Programming (cp-algorithms.com)](https://cp-algorithms.com/sequences/longest_increasing_subsequence.html) |
| [Longest common subsequence problem - Wikipedia](https://en.wikipedia.org/wiki/Longest_common_subsequence_problem) |
| [Longest Common Subsequence (tutorialspoint.com)](https://www.tutorialspoint.com/design_and_analysis_of_algorithms/design_and_analysis_of_algorithms_longest_common_subsequence.htm) |
| [Longest Common Subsequence Problem (techiedelight.com)](https://www.techiedelight.com/longest-common-subsequence/) |
| [4.9 Longest Common Subsequence (LCS) - Recursion and Dynamic Programming - YouTube](https://www.youtube.com/watch?v=sSno9rV8Rhg) |
| Dynamic Programming on Tree | [Dynamic Programming on Trees | Set-1 - GeeksforGeeks](https://www.geeksforgeeks.org/dynamic-programming-trees-set-1/) |
| [Dynamic Programming on Trees | Set 2 - GeeksforGeeks](https://www.geeksforgeeks.org/dynamic-programming-trees-set-2/) |
| Sparse Table | [Sparse Table - GeeksforGeeks](https://www.geeksforgeeks.org/sparse-table/?ref=lbp) |
| [Sparse Table - Algorithms for Competitive Programming (cp-algorithms.com)](https://cp-algorithms.com/data_structures/sparse-table.html) |
| Matrix Exponentiation | [Binary Exponentiation - Algorithms for Competitive Programming (cp-algorithms.com)](https://cp-algorithms.com/algebra/binary-exp.html) |
| [Matrix Exponentiation - GeeksforGeeks](https://www.geeksforgeeks.org/matrix-exponentiation/) |
| [Matrix exponentiation | HackerEarth](https://www.hackerearth.com/practice/notes/matrix-exponentiation-1/) |
| [Matrix exponential - Wikipedia](https://en.wikipedia.org/wiki/Matrix_exponential) |
| [Fibonacci Numbers in O(logn) [Matrix Exponentiation] - Only Code](https://www.onlycode.in/fibonacci-numbers-in-ologn-matrix-exponentiation/) |
| [Solving the Fibonacci Sequence with Matrix Exponentiation - YouTube](https://www.youtube.com/watch?v=EEb6JP3NXBI) |
| [What is Fast Exponentiation? - YouTube](https://www.youtube.com/watch?v=-3Lt-EwR_Hw&t=29s) |
| [Matrix Exponentiation · USACO Guide](https://usaco.guide/plat/matrix-expo?lang=cpp) |
| Dynamic Programming with Bitmask | [Bitmasking and Dynamic Programming | Set 1 (Count ways to assign unique cap to every person) - GeeksforGeeks](https://www.geeksforgeeks.org/bitmasking-and-dynamic-programming-set-1-count-ways-to-assign-unique-cap-to-every-person/?ref=rp) |
| [Bitmasking and Dynamic Programming | Set-2 (TSP) - GeeksforGeeks](https://www.geeksforgeeks.org/bitmasking-dynamic-programming-set-2-tsp/?ref=lbp) |
| Knuth’s Optimization | [Knuth's Optimization in Dynamic Programming - GeeksforGeeks](https://www.geeksforgeeks.org/knuths-optimization-in-dynamic-programming/) |
| [Knuth's Optimization - Algorithms for Competitive Programming (cp-algorithms.com)](https://cp-algorithms.com/dynamic_programming/knuth-optimization.html) |
| [Efficient dynamic programming using quadrangle inequalities | Proceedings of the twelfth annual ACM symposium on Theory of computing](https://dl.acm.org/doi/abs/10.1145/800141.804691) |
| [10003.pdf (onlinejudge.org)](https://onlinejudge.org/external/100/10003.pdf) (Cutting Sticks) |
| Khác | [Dynamic programming - Wikipedia](https://en.wikipedia.org/wiki/Dynamic_programming) |
| [5 Simple Steps for Solving Dynamic Programming Problems - YouTube](https://www.youtube.com/watch?v=aPQY__2H3tE) |
| [Dynamic Programming lectures - YouTube](https://www.youtube.com/playlist?list=PLl0KD3g-oDOGJUdmhFk19LaPgrfmAGQfo) |

# 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% |