SEASON 1

Dynamic Programming

Cơ bản về lập trình quy hoạch động

Khoa Tran Viet[1*]

[1]Department IT of Hue University, VietName

Correspondence
77 Nguyen Hue street, Hue, VietNam
Email: tvkhoa.husc@gmail.com

Funding information

Lập trình quy hoạch động là một trong những kỹ thuật cơ bản nhất của người lập trình. Rất nhiều bài toán có lời giải đẹp bằng kỹ thuật này. Đối với dân lập trình thi đấu đây được xem là một nội dung mà không thể bỏ qua, càng nghiên cứu, thực hành càng nhiều sẽ có cơ hội giải được nhiều bài toán hơn bằng kỹ thuật này. Ngoài ra đây cũng còn là một kỹ thuật giúp người học tập phân tích một vấn đề, bài toán theo phương pháp tổng quát hóa. Bài viết nằm nằm trong loạt bài về quy hoạch động, được viết để hổ trợ người học theo dạng một bài hướng dẫn (tutorial).

KEYWORDS
DP (dynamic programming), tabulation, memoization, top-down, bottom-Up, divide and conquer, tail recursive

1 | GIỚI THIỆU

Lập trình quy hoạch động là phương pháp giải quyết một bài toán phức tạp bằng cách chia nó thành một tập hợp các bài toán con đơn giản hơn. Đó chính là ý tưởng của chia để trị và kỹ thuật đệ quy chính là nòng cốt. Tuy nhiên, khác với đệ quy chia để trị đó là cách giải từng bài toán con một lần và lưu trữ các giải pháp của chúng bằng cấu trúc dữ liệu dựa trên bộ nhớ (array, list, map, set, v.v.). Mỗi giải pháp của chương trình con được lập chỉ mục theo một cách nào đó, thường dựa trên các giá trị của các tham số đầu vào của nó, để tạo điều kiện tra cứu. Vì vậy, lần tiếp theo xảy ra cùng một bài toán con, thay vì tính toán lại giải pháp của nó, người ta chỉ cần tra cứu giải pháp đã được tính toán trước đó, nhờ đó tiết kiệm thời gian tính toán. Kỹ thuật lưu trữ các giải pháp cho các bài toán con thay vì tính toán lại chúng được gọi là ghi nhớ.
Câu chuyện sau kể về cách giải thích "ghi nhớ" của một câu bé 4 tuổi.
Thầy giáo: Hỏi $S = 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1$ là bao nhiêu?
Cậu bé: Đếm, cộng và trả lời $S = 8$.

Thầy giáo: Viết tiếp +1 vào bên phải của công thức và hỏi tổng.
Cậu bé: trả lời bằng 9.
Thầy giáo: Sao tính nhanh vậy?
Cậu bé: Không cần tính lại chỉ cần lấy tổng cộng thêm 1 là ok.
Câu chuyện trên cho thấy lập trình quy hoạch động chỉ là một cách thú vị để nói 'công cụ ghi nhớ để tiết kiệm thời gian tính toán sau này".

2 | CÀI ĐẶT

Việc cài đặt thuật toán DP bằng hai cách sau:

- Dùng bảng (Tabulation), sử dụng cấu trúc bảng một hoặc nhiều chiều để lưu kết quả các bước giải và thiết kế từ dưới lên và rất phù hợp với bản chất của thuật toán DP. Cách này được sử dụng nhiều đối với người lập trình.
- Dùng Memoization, cũng sử dụng bộ nhớ để lưu trữ kết quả nhưng thiết kế lời gọi bài toán từ trên xuống, có thể xem là một phiên bản cải tiến của kỹ thuật đệ quy [1] nhưng gần với DP hơn.

2.1 | Dùng bảng

Sử dụng bảng để lưu dữ liệu cho bài toán nhỏ nhất, để từ đó xây dựng các bài toán lớn hơn, Theo đó, một thuật toán DP bằng bảng gồm ba bước sau:

- Định nghĩa bảng.
- Điền dữ liệu vào bảng dựa vào quan hệ đệ quy.
- Truy xuất bảng.

Xét ví dụ cho bài toán cổ điển sau:

Bài tập 1 Tìm số thứ n của dãy Fibonacci

Tìm số thứ n của dãy Fibonacci $F_n = F_{n-1} + F_{n-2}, F[0] = 0, F[1] = 1$.
Input
- Dòng duy nhất chứa số nguyên dương n là chỉ số của F_n thỏa $1 \le n \le 10^5$.
Output
- In ra số cần tìm, do số này lớn nên cần Modulo cho $10^9 + 7$.
Samples
Input
5
Output
5

[1] Đệ quy với kết quả trả về qua tham số hàm - Tail Recursive

Rất ngây thơ để giải bài toán này bằng công thức đệ quy được định nghĩa trước của bài toán, vì độ phức tạp tính toán của nó rơi vào hàm mũ $O(a^n), a < 2$ và nó chỉ chạy được với $n < 40$ trong một giây. Mã nguồn như sau:

```
int Fib(int n) {
  if(n<=1) return n;
  else return Fib(n-1)+Fib(n-2);
}
```

Bài toán này giải bằng kỹ thuật DP rất hợp lý, vì dùng bảng ta lưu được kết quả của các phần tử F_i và chỉ truy xuất một lần cho phép tính các phần tử lớn hơn. Mã được thiết kế theo ba bước như sau:

FiboTab.cpp

```
typedef long long LL;
const int mod = 1e9 + 7;
LL FiboTab(int n){
  // 1. Declare Tab
  vector <LL> tab(n,0);
  // 2. Fill data for Tab
  tab[0] = 0; tab[1] = 1;
  for (int i = 2; i < N; i++)
    tab[i] = (tab[i-1] + tab[i-2]) % mod;
  // 3. Get info from tab
  return tab[n];
}
```

Nhận xét:
+ Có thể khai báo mảng ở toàn cục ngoài hàm main() hoặc địa phương trong hàm main() cho mảng.
+ Luôn đảm bảo ba bước và ở bước thứ 2, quan trọng nhất làm tìm ra quy tắc (công thức truy hồi) để điền bảng.
+ Một số bài toán cần yêu câu xử lý truy vết, tức là không những lấy một mà nhiều giá trị của bảng.

2.2 | Dùng Memoization

Memoization tạm hiểu là được ghi nhớ. Kỹ thuật này ngược lại với kỹ thuật dùng bảng và vẫn sử dụng kỹ thuật gọi đệ quy như chính công thức ban đầu của thuật toán. Tuy nhiên, nó cũng cần một bảng để ghi kết quả của lời gọi đến hàm đệ quy và trả về kết quả ngay lập tức nếu gặp lời gọi đệ quy với tham số đưa vào đã giải ở lần trước.
Với bài toán tìm số Fibonacci, kết quả của câu gọi hàm sau bằng tổng kết quả của hai lần gọi hàm liền trước, dễ thấy ta có thể sử dụng memoization để tránh việc tính lại (đồng thời tránh luôn cả việc gọi đệ quy quá nhiều khiến vượt quá kích thước của stack). Mã cho bài toán trên như sau:

FiboMem.cpp

```cpp
#define NIL -1
#define MAX 100001
const int mod = 1e9 + 7;

vector<LL>lookup(MAX, NIL);
long long fibMem(int n){
  if (lookup[n] == NIL){
    if (n <= 1) lookup[n] = n;
    else lookup[n] = (fibMem(n-1) + fibMem(n-2))% mod;
  }
  return lookup[n];
}
```

Nhận xét:
+ Bảng phải khai báo toàn cục để hàm đệ quy tham chiếu.
+ Sử dụng được công thức đệ quy nên dễ viết hơn.

Để nắm bắt tốt hai kỹ thuật trên, xét thêm ví dụ cho bài toán cổ điển nữa như sau:

Bài tập 2 Tính tổ hợp không lặp chập k từ n phần tử

Cho công thức tổ hợp như sau: $\binom{n}{k} = \frac{n!}{k!(n-k)!}$. Lập trình tìm giá trị của công thức với k, n cho trước.
Input
- Dòng duy nhất chứa hai số nguyên dương k, n thỏa $0 \le k \le n \le 10^3$.
Output
- In ra số cần tìm.
Samples
Input
2 5
Output
10

Bài toán này có thể đưa vào bài toán giải quyết theo hai cách trên dựa vào công thức truy hồi còn gọi là tam giác Pascal sau:
$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}, \binom{n}{k} = 1$ if $n = k$ or $k = 0$.
Bảng lưu kết quả cho bài toán này là mảng hai chiều, mã nguồn DP cho nó như sau:

binomialDP.cpp

```cpp
int binomialCoefficient(int n, int k){
  int dp[n+1][k+1];
  int i,j;
  for(i=0;i<=n;i++)
    for(j=0;j<=i && j<=k;j++)
      if(j==0 || j==i) dp[i][j]=1;
      else
        dp[i][j]=dp[i-1][j-1]+dp[i-1][j];
  return dp[n][k];
}
```

Dựa vào công thức đệ quy, bản cài đặt bằng phương pháp Memoization như sau:

BinomialMemoi.cpp

```cpp
#include <bits/stdc++.h>
#define MAX_N 100
using namespace std;
int n, k;

vector<vector<int> >  Lookup(MAX_N,vector<int> (MAX_N,-1));

int BinoMemoi(int n, int k){
  if (k == n) return 1;
  if (k == 0) return 1;
  if (Lookup[n][k] !=-1) return Lookup[n][k];
  Lookup[n][k] = BinoMemoi(n-1,k) + BinoMemoi(n-1, k-1);
  return Lookup[n][k];
}
int main()
{
  cin >> n >> k;
  cout<<BinoMemoi(n,k)<<endl;
  return 0;
}
```

2.3 | Kết luận

Vậy kỹ thuật lập trình DP dùng để giải cho những dạng bài toán sau:

• Khử đệ quy của những bài toán đệ quy có độ phức tạp hàm mũ, thường là có bài toán con gối nhau (Overlapping Subproblems).

• Giải các bài toán tối ưu với nguyên lý bellman:" Một dãy các lựa chọn $a_1, a_2, \ldots, a_n$ là tối ưu thì các dãy con $a_i, \ldots, a_j$, với $1 \le i < j \le n$ của nó là tối ưu". Dựa trên nguyên lý tối ưu đó, giải bài toán theo phương pháp từ dưới lên với bảng kết quả của bài toán con và tối ưu ở mỗi bước giải.

Điểm mấu chốt của cách giải bài toán giải bằng DP là:

• Tìm được công thức toán học thể hiện bằng công thức đệ quy hay còn gọi là hệ thức truy hồi, phù hợp với cách thiết kế tổng quan cho bài toán.
• Chọn một trong hai kỹ thuật cài đặt là Tabulation hoặc Memoization để giải quyết.
• Có thể giải bằng thuật toán BruteForce để kiểm chứng.

3 | CÁC BÀI TOÁN DP

3.1 | DP cơ bản

3.1.1 | Dãy con liên tiếp có tổng lớn nhất

Bài tập 3 Largest Sum Contiguous Subsequence

Cho một dãy n số nguyên $a_1, a_2, \ldots, a_n$. Tìm dãy con $a_i, \ldots, a_j, 1 \le i \le j \le n$ sao cho tổng các phần tử trong dãy con là lớn nhất.
Input
- Dòng đầu tiên chứa số nguyên dương n là số phần tử của dãy thỏa $1 \le n \le 2.10^5$.
- Dòng thứ hai chứa n số nguyên a_i các phần tử cách nhau dấu cách thỏa $|a_i| \le 10^9$.
Output
- In ra tổng lớn nhất cần tìm (có thể yêu cầu in ra dãy con).
Samples
Input
5
2 4 6 8 10
Output
30
Ref: http://oj.hueuni.edu.vn/practice/problem/829/details

Nhận xét:
+ Nếu a là toàn số dương thì tổng các phần tử của a chính là kết quả.
+ Nếu a là toàn số âm thì số lớn nhất của các phần tử của a chính là kết quả.
+ Nếu a chỉ có một phần tử thì phần tử đó chính là kết quả cần tìm và đây được xem là trường hợp suy biến khi tìm công thức truy hồi cho bài toán.

Dựa vào nhận xét trên, gọi $S(n)$ là tổng cần tìm, dễ dàng ta thấy quan hệ hồi quy như sau:

$$S(n) = \begin{cases} a[1] & n = 1 \\ max(S(n-1) + a[n], a[n]) & n > 1 \end{cases}$$

Vậy phương án DP cho bài toán này gồm ba bước sau:

- Định nghĩa và khai báo mảng S, lấy chỉ số từ 1.
- Điền dữ liệu vào bảng: S[1]=A[1]; for i=2 to n do S[i] = max (S[i-1]+A[i], A[i])
- Truy xuất bảng với phần tử lớn nhất của mảng S.

Xét ví dụ $A = \{-6, 2, -4, 1, 3, -1, 5, -1\}$, bằng việc điền dữ liệu theo quy tắc trên ta có mảng S tương ứng: $S = \{-6, 2, -2, 1, 4, 3, 8, 7\}$. Vậy kết quả bằng 8 và dãy con liên tiếp là $\{1, 3, -1, 5\}$.

LSCS.cpp

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef long long int lli;
typedef vector<lli> vi;

int main() {
  ios_base::sync_with_stdio(0); cin.tie(0); cout.tie(0);
  int n; cin >> n;
  vi A(n+1), S(n+1);
  for (int i = 1; i <= n; i++) cin >> A[i];
  S[1]=A[1];lli ans = S[1];
  for (int i = 2; i <= n; i++) {
    S[i] = max(S[i-1] + A[i], A[i]);
    ans = max(ans, S[i]);
  }
  cout << ans << endl;
  return 0;
}
```

Thách thức

+ Hãy in dãy con cần tìm, một thao tác truy xuất cần thiết khi lập trình DP.

Bài này cũng có thể giải bằng cách cài đặt Memoization. Cấu trúc bảng được đề xuất cho kỹ thuật Memoization là từ điển (Map in STL), thuận tiện cho việc dò tìm vết. Ngoài ra, một thuật toán hay đề xuất cho bài toán này có tên là Kadane [2].

[2] https://algorithms.tutorialhorizon.com/kadanes-algorithm-maximum-subarray-problem/

LSCSMemoi.cpp

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef long long int lli;
typedef vector<lli> vi;
const long long int NIL = -1;
vi A(200002), Lookup(200002,-1);
lli ans;

lli S(int n){
  if (Lookup[n]==NIL){
    if (n==1) Lookup[n]=A[n];ans=A[n];
    else{
      Lookup[n]= max(S(n-1)+A[n], A[n]);
      ans=max(ans, Lookup[n]);
    }
  }
  return Lookup[n];
}
int main()
{
  ios_base::sync_with_stdio(0); cin.tie(0); cout.tie(0);
  int n; cin >> n;
  for (int i = 1; i <= n; i++) cin >> A[i];
  lli r=S(n);
  cout<<ans<<endl;
  return 0;
}
```

Thách thức

Hãy sử dụng cấu trúc pair hoặc map, unorder map để gộp hai phần tử là Lookup và ans thành một cặp và trả về cho hàm, sau đó truy cập phần tử thứ hai ans cho kết quả của bài toán.

3.1.2 | Bài toán cái túi

Bài tập 4 0 1 Knapsack

Cho n vật phẩm đánh chỉ số từ 1 đến n, mỗi vật phẩm có hai giá trị là trọng lượng w_i và giá tiền v_i. Bạn có quyền chọn các vật phẩm từ n vật phẩm trên để vào trong ba lô của bạn. Biết rằng ba lô của bạn có sức chứa với tổng trọng lượng là C. Hãy lập trình tính tổng giá trị lớn nhất có thể của các vật phẩm mà bạn nhét vào ba lô của mình.

Input

- Dòng đầu tiên chứa số nguyên dương n, C là số phần tử vật phẩm và sức chứa của ba lô thỏa $1 \le n \le 100, 0 \le C \le 10^5$.
- n dòng tiếp theo mỗi dòng chứa hai số wi, v_i lần lượt là trọng lượng và giá trị của vật phẩm thứ i thỏa $0 \le v_i \le 50, 0 \le w_i \le 10^5$.

Output

- In ra tổng lớn nhất cần tìm. (Có thể yêu cầu in ra các phần tử được chọn).

Samples

Input

3 8

3 30

4 50

5 60

Output

90

Note: Phần tử 1 và 3 được chọn, trọng lương = 3+ 5 = 8 thỏa mãn sức chứa và tổng tiền= 30 + 60 = 90.

https://atcoder.jp/contests/dp/tasks/dp_d

Nhận xét:

+ Bài này vốn là bài toán của thuật toán tham lam (greedy algorithm), tuy nhiên nó cũng được giải một cách dễ dàng bằng DP.

+ 0/1 có nghĩa ta có thể chọn hoặc không chọn vật phẩm đặt vào ba lô, như vậy danh sách các vật phẩm được chọn hay không như một xâu nhị phân.

Lời giải:

+ Gọi $x_1, x_2, \ldots, x_n$ là các biến thể hiện sự chọn lựa của các vật phẩm bỏ vào ba lô. Ta có, tổng trọng lượng của các vật thể nếu chọn được hết là $w_1x_1 + w_2x_2 + \ldots w_nx_n$ và tổng giá trị vật phẩm có thể là $v_1x_1 + v_2x_2 + \cdots + v_nx_n$.

+ Ví dụ: Cho các vật phẩm và ba lô có sức chứa trọng lượng $C = 20$.

$item_1 = (w_1 = 2, v_1 = 3)$.

$item_1 = (w_2 = 4, v_2 = 5)$.

$item_1 = (w_3 = 5, v_3 = 6)$.

$item_1 = (w_4 = 7, v_4 = 8)$.

$item_1 = (w_5 = 9, v_5 = 10)$.

Kết quả lựa chọn ta có $(x_1, x_2, x_3, x_4, x_5) = (1, 1, 1, 0, 1)$ và tổng giá trị=24.

+ Gọi $KS(n, C)$ là kết quả của bài toán với việc bỏ vật phẩm n vào ba lô với sức chứa c, ta có: $KS(n, C) = max(v_1x_1 + v_2x_2 + \cdots + v_nx_n)$ thỏa $w_1x_1 + w_2x_2 + \ldots w_nx_n \leq c, x_i = 0, 1$.
+ Với giới hạn sức chứa (trọng lượng) C, việc chọn tối ưu trong số các vật phẩm từ 1 đến n để có giá trị lớn nhất với hai khả năng sau:

- Nếu không chọn vật phẩm thứ n thì nhờ người giúp đỡ [3] giải bài toán với chọn phần tử thứ $n-1$ có cùng trọng lượng, nói cách khác $KS(n, C) = KS(n-1, C)$.
- Nếu không chọn vật phẩm thứ n và chỉ xét tới trường hợp này khi $C > w[n]$ thì nhờ người giúp đỡ giải bài toán chọn phần tử thứ $n-1$ nhưng với ba lô có kích thước nhỏ hơn ($C - w[n]$) và cộng thêm giá trị $v[n]$, nói cách khác $KS(n, C) = v[n] + KS(n-1, C-w[n])$.
- Do lựa chọn tối ưu nên chỉ chọn một trong hai cách trên (01) nên $KS(n, C)$ sẽ là giá trị lớn nhất một trong hai giá trị tương ứng hai cách chọn trên.

Dựa vào đó ta có công thức truy hồi, và lời giải bằng cách Memoization cho công thức như sau;

KnapsackMemoi.cpp

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef long long int lli;
#define MAX_N  1001
#define MAX_W  1000001
int N, C;

vector<vector<lli> >  Lookup(MAX_N,vector<lli> (MAX_W,-1));
vector<int> wei(MAX_N);
vector<lli> val(MAX_N);

lli Knapsack(int n,int  c){
  if (Lookup[n][c] == -1){
    if(c==0|| n==0)return 0;
    else if(c-wei[n]>=0)
      return Lookup[n][c] = max(Knapsack(n-1, c-wei[n])+val[n], Knapsack(n-1, c));
  else
    return Knapsack(n-1, c);
  }
  return Lookup[n][c];
}
int main()
{
  cin>>N>>C;
  for (int i =1;i<=N;i++) cin>>wei[i]>>val[i];
  cout<<Knapsack(N, C)<<endl;
  return 0;
}
```

[3] Người phương tây viết là người giúp đỡ (helper).

Với bảng Lookup cho phương pháp Memoization, ta dễ dàng triển khai cho phương pháp bảng như sau:

KnapsackDP.cpp

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef long long int lli;

int main()
  {
  ios_base::sync_with_stdio(false); cin.tie(NULL);
  lli N, W; cin >> N >> W;
  vector <lli> wei(N+1), val(N+1);
  for (lli i=1; i<=N;++i) cin >> wei[i] >> val[i];

  vector < vector<lli> > Lookup(N+1, vector <lli> (W+1));

  for (lli i=0; i<=W;++i) Lookup[0][i]=0;
  for (lli i=1; i<=N;++i) {
    for (lli j=1; j<=W;++j){
      if (wei[i]>j) Lookup[i][j]=Lookup[i-1][j];
      else Lookup[i][j] = max(Lookup[i-1][j],
      Lookup[i-1][j-wei[i]] + val[i]);
    }
  }

  cout << Lookup[N][W] << "\n";
  return 0;
}
```

Nhận xét và thách thức:
+ Một trong những kỹ thuật của việc điền bảng là xử lý biên (cột, dòng biên) khi truy xuất với quan hệ đệ quy trạng thái thứ i gọi trạng thái $i+1$ hoặc $i-1$.
+ Truy vết để chỉ ra vật phẩm nào được chọn.
+ Nộp bài trên oj link: http://oj.hueuni.edu.vn/practice/problem/848/details.

Rất nhiều bài toán DP cần xử lý trên dữ liệu nhiều chiều, trong trường hợp này việc định nghĩa quan hệ truy hồi cũng như phương án xây dựng bảng đòi hỏi người lập trình có tư duy đa chiều và các bài toán này thường khó hơn. Dạng này thường có nhiều biến phụ thuộc, do vậy bài toán sẽ phát triển theo nhiều chiều.
Một số bài toán cổ điển sau cần nắm bắt.

3.1.3 | Biến đổi xâu

Trong xử lý xâu, phép đo sự tương đồng là bài toán thường xuyên xảy ra. Hoặc là biến đổi xâu này thành xâu khác sao cho dùng ít phép biến đổi nhất. Bài toán sau là một dạng trên được gọi là bài toán Edit distance

xác định số phép toán ít nhất gồm (thêm, xóa, sửa ký tự) để biến đổi xâu này thành xâu khác. Bài toán gốc là khoảng cách Levenshtein tên của một nhà khoa học máy tính Xô Viết.

Bài tập 5 Edit distance

Số phép biến đổi giữa hai xâu là số lượng thao tác tối thiểu cần thiết để chuyển đổi một chuỗi thành chuỗi khác. Các thao tác được phép là:
- Thêm một ký tự vào chuỗi.
- Xóa một ký tự khỏi chuỗi.
- Thay thế một ký tự trong chuỗi.

Nhiệm vụ của bạn là tính số phép biến đổi giữa hai xâu.
Input
- Dòng thứ nhất chứa xâu thứ nhất gồm m ký tự in hoa.
- Dòng thứ hai chứa xâu thứ hai gồm n ký tự in hoa thỏa $1 \le n, m \le 5000$.

Output
- In số phép biến đổi cần tìm

Samples
Input
LOVE
MOVIE
Output
2
Ref: http://oj.hueuni.edu.vn/practice/problem/834/details

Lời giải:
Xét ví dụ chuyển giữa hai xâu là kitten và sitting, ta có thể thực hiện ba phép biến đổi như sau:
i) kitten -> sitten (thay 'k' bằng 's')
ii) sitten -> sittin (thay 'e' bằng 'i').
iii) sittin -> sitting (thêm 'g' vào cuối).
Gọi $X = [1..m]$ và $Y[1..n]$ là hai xâu dữ liệu vào, nhận thấy:
Trường hợp 1. Suy biến.
- Nếu $m = 0$ ta mất $cost = n$ phép toán để có hai xâu bằng nhau, ví dụ: ("", "ABC")->("ABC", "ABC").
- Nếu $n = 0$ thì ngược lại mất $cost = m$ phép toán, ví dụ: ("ABCD", " ")-> ("ABCD", "ABCD").

Bài toán này cũng thuộc dạng tối ưu bài toán gối nhau (substructure), cho nên định nghĩa đệ quy sau:
Trường hợp 2.
- Nếu ký tự cuối của X và Y bằng nhau thì mất $cost = 0$ vì ta không xét ký tự cuối mà chỉ xét bài toán con kế tiếp cho $X[1..m-1]$ và $y[1..n-1]$.

Trường hợp 3. Nếu ký tự cuối của X và Y khác nhau thì giá trị $cost$ tối thiểu có được bằng 3 thao tác sau:
a). Chèn ký tự cuối của Y vào X (cost=1) và ta có kết quả X giữ nguyên còn Y giảm đi 1 ký tự cuối, ví dụ: ("ABA", "ABC")->("ABAC", "ABC")->("ABA", "AB") theo trường hợp 2.
b). Ngược với thao tác chèn là xóa và ta xóa ký tự cuối của X (cost=1), kết quả Y giữ nguyên và X giảm đi ký tự cuối, ví dụ: ("ABA", "ABC")-> ("AB", "ABC").

c). Thay thế ký tự cuối của X bởi ký tự cuối của Y (cost=1) và kết quả X, Y đều giảm một đơn vị, ví dụ: ("ABA", "ABC")->("ABC", "ABC")-> ("AB", "AB") theo trường hợp 2.

Từ khai thác các tính chất trên, ta có công thức truy hồi sau:

$$dist[m][n] = \begin{cases} max(m,n), & \text{khi min(m, n) =0} \\ dist[m-1][n-1], & \text{khi } X[m] = Y[n] \\ 1 + min(dist[m-1][n], dist[m][n-1], dist[m-1][n-1]), & \text{khi } X[m] \neq Y[n] \end{cases}$$

Thuật toán cài đặt bằng dùng bảng (Tabulation) như sau:

EditDistance.cpp

```cpp
#include <iostream>
using namespace std;

int dist(string X, int m, string Y, int n){
  int T[m + 1][n + 1] = { 0 };
  for (int i = 1; i <= m; i++)
    T[i][0] = i;                // (case 1)
  for (int j = 1; j <= n; j++)
    T[0][j] = j;                // (case 1)
  int substitutionCost;
  //fill the lookup table in bottom-up manner
  for (int i = 1; i <= m; i++){
    for (int j = 1; j <= n; j++){
      if (X[i] == Y[j])               // (case 2)
        substitutionCost = 0;        // (case 2)
      else
        substitutionCost = 1;        // (case 3c)

      T[i][j] = min(min(T[i - 1][j] + 1,   // deletion (case 3b)
        T[i][j - 1] + 1),        // insertion (case 3a)
        T[i - 1][j - 1] + substitutionCost);
                  // replace (case 2 & 3c)
    }
  }
  return T[m][n];
}
int main()
{
  string X; cin>>X; X=' '+X;
  string Y; cin>>Y; Y=' '+Y;
  cout << dist(X, X.length(), Y, Y.length());
  return 0;
}
```

Thách thức:
+ Giải bài toán bằng phương pháp Memoization.

3.1.4 | Xâu con chung dài nhất

Bài tập 6 Longest Common Subsequence

Xâu ký tự X được gọi là xâu con của xâu ký tự Y nếu ta có thể xoá đi một số ký tự trong xâu Y để được xâu X.
Cho biết hai xâu ký tự A và B độ dài không quá 3000 ký tự in thường. Hãy tìm xâu ký tự C có độ dài lớn nhất và là con của cả A và B.
Input
- Dòng thứ nhất chứa xâu thứ nhất gồm n ký tự in thường.
- Dòng thứ hai chứa xâu thứ hai gồm m ký tự in thường thỏa $1 \leq n, m \leq 5000$.
Output
- In ra độ dài của xâu con cần tìm.
Samples
Input
LOVE
MOVIE
Output
3
Ref: https://atcoder.jp/contests/dp/tasks/dp_f

Lời giải:
Xét hai xâu $X = ABCBDAB$, xâu $Y = BDCABA$. Xâu chung con dài nhất của X và Y là 4 với các xâu có thể như sau: BDAB, BCAB và BCBA. Như vậy, bài toán này thuộc dạng nhiều kết quả nếu muốn in một xâu con chung dài nhất.
Cũng thuộc dạng bài toán tối ưu bài toán gối nhau, nên phương án nghĩ tới giải nó chính là tìm quan hệ truy hồi. Nếu có được công thức, kỹ thuật dùng bảng hay memoization sẽ giúp ta giải quyết vấn đề.
Gọi $X = [1..m]$ và $Y[1..n]$ là hai xâu dữ liệu vào, nhận thấy:
- Trường hợp 1. Cả hai xâu đều kết thúc với ký tự giống nhau, ta giải bài toán con với kích thước giảm đi một trên cả hai xâu và kết quả xâu con là nối thêm ký tự cuối đó vào, ví dụ: LSC("MOVIE", "LOVE") -> LSC("MOVI", "LOV")+ 'E'. Nếu không muốn lấy xâu con thì + 1 cho trường hợp tính độ dài.
- Trường hợp 2. Cả hai xâu không giống nhau ký tự cuối, ví dụ: $X = ABCBDAB, m = 7$, $Y = BDCABA, n = 6$. Xâu chung dài nhất sẽ là max(LSC(X[1..m-1], Y[1..n]), LSC(X[1..m], Y[1..n-1]).
Vấn đề đặt ra là LSC sẽ kết thúc như thế nào? Đơn giản chính là lúc m=0 hoặc n=0.
Ví dụ quy trình đệ quy cho hai xâu X, Y trên:
LCS("ABCBDAB", "BDCABA") = maximum (LCS("ABCBDA", "BDCABA"), LCS("ABCBDAB", "BDCAB"))
LCS("ABCBDA", "BDCABA") = LCS("ABCBD", "BDCAB") + "A"

LCS("ABCBDAB", "BDCAB") = LCS("ABCBDA", "BDCA") + "B"
LCS("ABCBD", "BDCAB") = maximum (LCS("ABCB", "BDCAB"), LCS("ABCBD", "BDCA"))
LCS("ABCBDA", "BDCA") = LCS("ABCBD", "BDC") + "A"
và cứ như vậy cho đến khi gặp suy biến.
Vậy ta có công thức truy hồi sau:

$$LSC[m][n] = \begin{cases} 0, & \text{khi } m = n = 0 \\ LSC[m-1][n-1] + 1, & \text{khi } X[m] = Y[n] \\ maximum(LSC[m-1][n], LSC[m][n-1]), & \text{khi } X[m] \neq Y[n] \end{cases}$$

Độ phức tạp của thuật toán đệ quy trên là $O(2^{n+m})$ nếu cài bằng đệ quy. Thuật toán cài đặt bằng dùng bảng (Tabulation) như sau với độ phức tạp chỉ còn là $O(n \times m)$:

LSC.cpp

```cpp
#include <bits/stdc++.h>
using namespace std;
int lcs(string &a, string &b) {
  int m = a.size(), n = b.size();
  a = ' ' + a;
  b = ' ' + b;
  vector< vector<int> > f(m+1, vector<int>(n+1, 0));
  for (int i=1; i<=m; i++)
    for (int j=1; j<=n; j++) {
      if (a[i] == b[j]) f[i][j] = f[i-1][j-1] + 1;
      else f[i][j] = max(f[i-1][j], f[i][j-1]);
    }
    return f[m][n];
}
int main() {
  string a; cin >> a;
  string b; cin >> b;
  cout << lcs(a, b);
  return 0;
}
```

Thách thức và thông tin:
+ Giải bài toán bằng phương pháp Memoization.
+ Liệt kê hết các xâu con theo thứ tự từ điển.
+ Giải và nộp bài tại: https://atcoder.jp/contests/dp/tasks/dp_f
+ Đây chỉ là bài toán gốc, các biến thể của bài toán LSC phức tạp hơn nhiều đòi hỏi rất nhiều kỹ thuật cũng như việc phân tích để đưa ra cách tiếp cận bài toán một cách tối ưu. Tham khảo https://vnoi.info/forum/5/4953/

4 | TỔNG KẾT

Bài viết chỉ khai thác về mặt tiếp cận giải bài toán theo phương pháp truyền thống là sự tổng quát hóa thể hiện bằng việc tìm hệ thức truy hồi và dùng hai kỹ thuật chính để giải bài toán quy hoạch động là dùng bảng (Tabulation) và dùng Memoization.

Nội dung được trình bày và tổng hợp từ nhiều nguồn tài liệu trên internet, chủ yếu từ kiến thức của lập trình thi đấu (competitive prgramming) do đó có thể không tổng quát. Văn phong theo chủ quan của người viết, do đó có nhiều sai sót. Mọi đóng góp cho bài viết hay hơn, chính xác hơn các bạn gửi về email: tvkhoa.husc@gmail.com. Chúc vui và hẹn gặp ở season 2.