Quy Hoạch Động

# (Dynamic Programming)

1/1/2012

Hồ Đắc Phương

# Dynamic Programming – Quy hoạch động.

DP là chiến lược giải bài toán phức tạp bằng cách chia nhỏ vấn đề to thành nhiều vấn đề con, và lưu lại kết quả tính toán các vấn đề con để tránh việc tính toán trùng lặp. Các bài toán giải được theo phương pháp này có hai đặc tính:

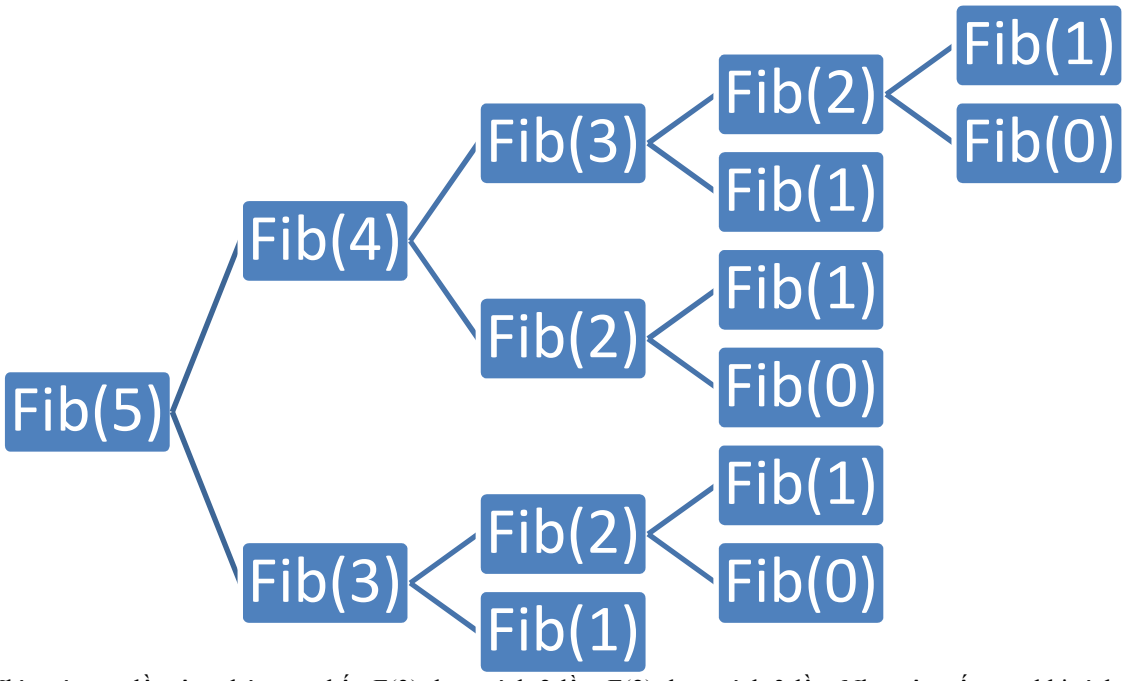
(1) Các bài toán con chồng chéo lên nhau
(2) Có cấu trúc tối ưu

**Chồng Chéo**

Giống D&C, DP kết hợp kết quả của các bài toán con lại. DP được sử dụng khi kết quả của các bài toán con được sử dụng đi sử dụng lại. Trong DP, kết quả bài toán con được lưu giữ lại (có thể đặt trong bảng) để tránh việc tính toán lại. Như vậy, trong trường hợp các bài toán con không chồng chéo với nhau thì DP không cần thiết (không cần phải lưu lại các kết quả con vì chúng không còn được dùng tới). Rõ ràng tìm kiếm nhị phân thuộc loại này. Tuy nhiên, hãy xét chương trình tìm số Fibonacci sau đây, trong chương trình có nhiều bài toán con được tính toán trùng lặp nhiều lần.

```
/* Chương trình con tính số Fibonacci thứ n */
int fib(int n)
{
  if ( n <= 1 ) return n;
  return fib(n-1) + fib(n-2);
}
```

Khi tính fib(5), chúng ta sẽ nhìn thấy việc tính toán trùng lặp như sau:




Nhìn vào sơ đồ trên, chúng ta thấy F(3) được tính 2 lần, F(2) được tính 3 lần. Như vậy nếu sau khi tính toán ở lần đầu tiên, lưu lại giá trị F(3) hay F(2) thì ở những lần sau, chúng ta không cần phải tính toán lặp lại nữa. Ở đây chúng ta có hai phương pháp lưu giữ:

(1) Ghi nhớ (Memorization - Top Down)
(2) Lập bảng (Tabulation - Bottom-Up)

**Ghi Nhớ**. Đây là biến tấu của đệ quy, tăng tốc độ đệ quy, tránh việc lặp lại các tính toán dư thừa bằng cách sử dụng bảng tìm kiếm (lookup). Ban đầu, khởi tạo toàn bộ các phần tử trong bảng lookup bằng giá trị NIL. Khi cần kết quả một bài toán con n, đầu tiên chúng ta tra bảng lookup. Nếu đã có kết quả trong bảng, chúng ta chỉ việc sử dụng. Nếu chưa có kết quả, chúng ta tìm và lưu kết quả tại vị trí n của bảng lookup (để dùng cho những lần sau).

```
/* Tính toán dãy số Fib kiểu ghi nhớ */
#include<stdio.h>
#define NIL -1
#define MAX 100

int lookup[MAX];

/* Khởi tạo NIL cho bảng lookup */
void _initialize()
{
  int i;
  for (i = 0; i < MAX; i++)
    lookup[i] = NIL;
}

/* function for nth Fibonacci number */
int fib(int n)
{
   if(lookup[n] == NIL)
   {
    if ( n <= 1 )
      lookup[n] = n;
    else
      lookup[n] = fib(n-1) + fib(n-2);
   }

   return lookup[n];
}

int main ()
{
  int n = 40;
  _initialize();
  printf("Fibonacci number is %d ", fib(n));
  getchar();
  return 0;
}
```

**Lập Bảng**. Chương trình tính toán bảng giá trị cho các bài toán con từ dưới lên, từ những bài toán nhỏ nhất đến bài toán lớn cuối cùng.

```
/* Tính toán dãy số Fib kiểu lập bảng */
#include<stdio.h>
int fib(int n)
{
  int f[n+1];
  int i;
  f[0] = 0;   f[1] = 1;
  for (i = 2; i <= n; i++)
      f[i] = f[i-1] + f[i-2];

  return f[n];
```

```
}

int main ()
{
  int n = 9;
  printf("Fibonacci number is %d ", fib(n));
  getchar();
  return 0;
}
```

Cả phương pháp Ghi Nhớ và Lập Bảng đều lưu lại kết quả các bài toán con. Trong phương pháp Ghi Nhớ, giá trị các phần tử bảng được tính toán theo yêu cầu; trong khi ở phương pháp Lập Bảng, các phần tử trong bảng được tính toán lần lượt từ phần tử đầu tiên. Lưu ý là chưa chắc mọi phần tử của mảng lookup đã có đủ giá trị.

## 1. CHUỖI CON TĂNG DÀI NHẤT (LIS - Longest Increasing Subsequence)

Cho một chuỗi cho trước, hãy tìm độ dài lớn nhất của chuỗi con tăng (là chuỗi con mà các phần tử được sắp xếp theo thứ tự tăng dần. Ví dụ, độ dài của LIS của dãy { 10, 22, 9, 33, 21, 50, 41, 60, 80 } là 6 và LIS là {10, 22, 33, 50, 60, 80}.

**Cấu trúc con tối ưu:** Giả sử chúng ta phải tìm LIS trong mảng arr[0..n-1]. Gọi L(i) là độ dài LIS cho đến chỉ số i sao cho arr[i] là một phần (và là phần tử cuối cùng) của LIS. Khi đó,

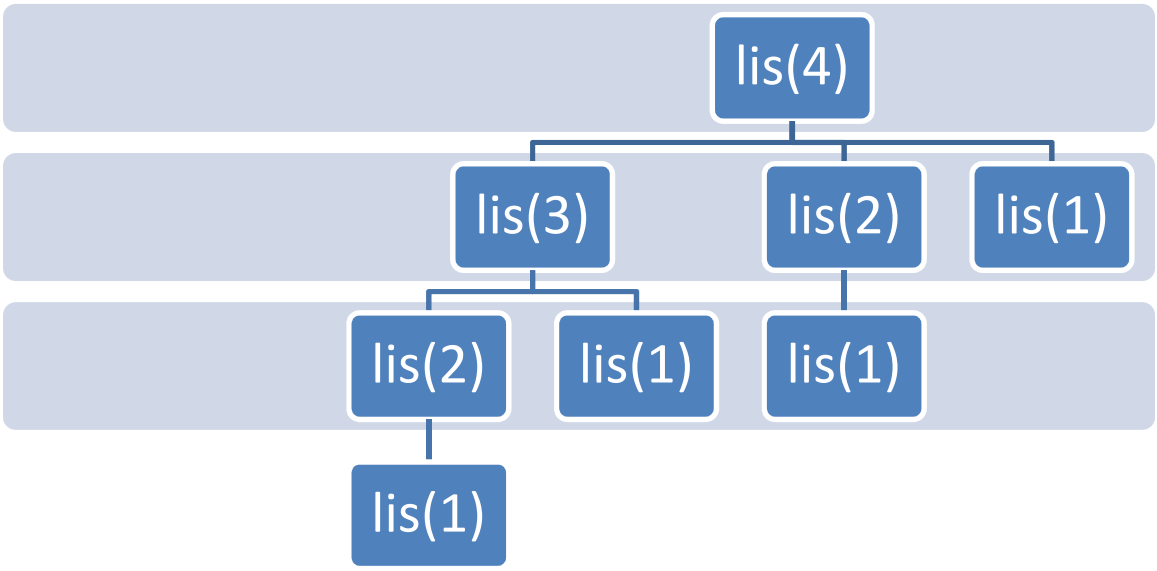
$LIS(i) = \{ 1 + Max_{(j< i \text{ và } arr[j] < arr[i])} ( LIS(j) ) \}$

= 1 khi không có j như vậy.

**Cách 1**. Giải pháp đệ quy đơn giản. Hàm trả lại hai giá trị: Độ dài của LIS và chỉ số của phần tử cuối cùng trong LIS.

```
#include<stdio.h>
#include<stdlib.h>

struct result
{
   int lis_len;
   int last_index;
};

struct result lis( int arr[], int n )
{
   struct result res, max;
   max.lis_len = 1;
   max.last_index = n-1;

   /* Trường hợp cơ bản */
   if(n == 1)
      return max;

   /* Đệ quy liên tiếp để nhận giá trị LIS của các phần tử bên trái */
   for(int i = 1; i < n; i++)
```

```
    {
        res = lis(arr, i);
        if (arr[res.last_index] < arr[n-1])
        {
            res.lis_len++;
            res.last_index = n - 1;
        }

        /* Cập nhật max khi cần thiết */
        if(res.lis_len  > max.lis_len)
            max = res;
    }
    return max;
}

int main()
{
    int arr[] = { 10, 22, 9, 33, 21, 50, 41, 60 };
    int n = sizeof(arr)/sizeof(arr[0]);
    struct result max = lis( arr, n );
    printf("Length of LIS is %d\n",  max.lis_len);

    getchar();
    return 0;
}
```




Quan sát cài đặt trên, chúng ta có thể nhìn thấy cây đệ quy, và nhiều bài toán con bị tính toán trùng lặp. lis(n) là độ  dài LIS cho mảng arr[]. Do đó giải pháp DP ở  đây có thể áp dụng được.

```
#include<stdio.h>
#include<stdlib.h>

/* lis() returns the length of the longest increasing subsequence in
   arr[] of size n */
int lis( int arr[], int n )
```

```
{
  int *lis, i, j, max = 0;
  lis = (int*) malloc ( sizeof( int ) * n );

  /* Initialize LIS values for all indexes */
  for ( i = 0; i < n; i++ )
    lis[i] = 1;

  /* Compute optimized LIS values in bottom up manner */
  for ( i = 1; i < n; i++ )
    for ( j = 0; j < i; j++ )
      if ( arr[i] > arr[j] && lis[i] < lis[j] + 1)
        lis[i] = lis[j] + 1;

  /* Pick maximum of all LIS values */
  for ( i = 0; i < n; i++ )
    if ( max < lis[i] )
      max = lis[i];

  /* Free memory to avoid memory leak */
  free( lis );

  return max;
}

/* Driver program to test above function */
int main()
{
  int arr[] = { 10, 22, 9, 33, 21, 50, 41, 60 };
  int n = sizeof(arr)/sizeof(arr[0]);
  printf("Length of LIS is %d\n", lis( arr, n ) );

  getchar();
  return 0;
}
```

Chú ý rằng, độ phức tạp của thuật toán này là $O(n_2)$, trong khi đó có thuật toán khác có độ phức tạp O(nlogn).

# CHUỖI CON CHUNG LỚN NHẤT (LCS – Longest Common Subsequence)

Bài toán xác định chuỗi con chung lớn nhất là ví dụ điển hình trong việc áp dụng DP : hãy tìm chuỗi con dài nhất nằm trong hai chuỗi cho trước. Chú ý các phần tử liên tiếp trong chuỗi con không nhất định phải nằm cạnh nhau trong chuỗi mẹ. Ví dụ, “abc”, “abg”, “bdf”, “aeg”, ‘”acefg”, .. đều là chuỗi con của “abcdefg”. Xâu độ dài n có 2n chuỗi con khác nhau.

Ví dụ :
LCS cho hai chuỗi “ABCDGH” và “AEDFHR” là “ADH” – độ dài 3.
LCS cho hai chuỗi “AGGTAB” và “GXTXAYB” là “GTAB” – độ dài 4.

Cách giải đơn giản nhất là sinh ra tất cả các chuỗi con của cả hai chuỗi ban đầu, và sau đó tìm chuỗi con dài nhất. Tuy nhiên chúng ta để ý:

Giả sử hai chuỗi ban đầu là X[0..m-1] và Y[0..n-1] tương ứng có độ dài m và n. Giả sử L(X[0..m-1], Y[0..n-1]) là độ dài LCS của hai chuỗi X, Y. Dưới đây là định nghĩa kiểu đệ quy của L(X[0..m-1], Y[0..n-1]).

Nếu hai chuỗi có phần tử cuối cùng giống nhau (tức là X[m-1] == Y[n-1]) thì: L(X[0..m-1], Y[0..n-1]) = 1 + L(X[0..m-2], Y[0..n-2])

Nếu không (tức là X[m-1] != Y[n-1]) thì L(X[0..m-1], Y[0..n-1]) = MAX ( L(X[0..m-2], Y[0..n-1]), L(X[0..m-1], Y[0..n-2])

Ví dụ:
1) Xét hai chuỗi "AGGTAB" và "GXTXAYB". Phần tử cuối cùng của hai chuỗi giống nhau, do đó: L("AGGTAB","GXTXAYB") = 1 + L("AGGTA","GXTXAY")

2) Xét hai chuỗi "ABCDGH" và "AEDFHR. Phần tử cuối cùng của hai chuỗi khác nhau, do đó:: L("ABCDGH","AEDFHR") = MAX ( L("ABCDG", "AEDFH**R**"), L("ABCDG**H**", "AEDFH") )

Vì có cấu trúc con tối ưu, nên kết quả bài toán lớn có thể được tổng hợp từ kết quả các bài toán con.

Dưới đây là cách giải đệ quy bài toán LCS theo đúng công thức nêu trên.

```
#include<stdio.h>
#include<stdlib.h>

int max(int a, int b);

/* Returns length of LCS for X[0..m-1], Y[0..n-1] */
int lcs( char *X, char *Y, int m, int n )
{
   if (m == 0 || n == 0)
     return 0;
   if(X[m-1] == Y[n-1])
     return 1 + lcs(X, Y, m-1, n-1);
   else
     return max(lcs(X, Y, m, n-1), lcs(X, Y, m-1, n));
}

/* Utility function to get max of 2 integers */
int max(int a, int b)
{
    return (a > b)? a : b;
}

int main()
{
  char X[] = "AGGTAB";
  char Y[] = "GXTXAYB";
```

```
  int m = strlen(X);
  int n = strlen(Y);

  printf("Length of LCS is %d\n", lcs( X, Y, m, n ) );

  getchar();
  return 0;
}
```

Giải pháp đệ quy sẽ bị tính toán trùng lặp rất nhiều, và dưới đây là giải pháp quy hoạch động:

```
#include<stdio.h>
#include<stdlib.h>

int max(int a, int b);

/* Returns length of LCS for X[0..m-1], Y[0..n-1] */
int lcs( char *X, char *Y, int m, int n )
{
   int L[m+1][n+1];
   int i, j;

   /* Following steps build L[m+1][n+1] in bottom up fashion. Note
      that L[i][j] contains length of LCS of X[0..i-1] and Y[0..j-1] */
   for (i=0; i<=m; i++)
   {
     for (j=0; j<=n; j++)
     {
       if (i == 0 || j == 0)
         L[i][j] = 0;

       else if(X[i-1] == Y[j-1])
         L[i][j] = L[i-1][j-1] + 1;

       else
         L[i][j] = max(L[i-1][j], L[i][j-1]);
     }
   }

   /* L[m][n] contains length of LCS for X[0..n-1] and Y[0..m-1] */
   return L[m][n];
}

int max(int a, int b)
{
    return (a > b)? a : b;
}

int main()
{
  char X[] = "AGGTAB";
  char Y[] = "GXTXAYB";
  int m = strlen(X);
  int n = strlen(Y);
```

```
  printf("Length of LCS is %d\n", lcs( X, Y, m, n ) );

  getchar();
  return 0;
}
```

# SỐ BƯỚC SỬA (Edit Distance)

Cho hai xâu kích thước m, n và tập hợp các thao tác: replace (R), insert (I) và delete (D). Tìm số lượng thao tác tối thiểu để biến xâu này thành xâu kia.

**Xác định phương thức đệ quy:**

Đâu là bài toán con? Hãy xét bài toán trên 2 xâu con, chẳng hạn hai xâu ***tiền tố (prefix)*** [1...i] và [1...j] với 1< i < m và 1 < j < n. Giả sử số bước sửa trong trường hợp này là E(i, j). Ở đây ta phải tìm E(m, n) với số thao tác tối thiểu. Trong hai từ tiền tố, chúng ta căn phải hai chuỗi theo 3 cách, (i, -), (-, j) và (i, j). Dấu trừ (-) thể hiện không có ký tự. Xét ví dụ sau: xét hai chuỗi SUNDAY và SATURDAY. Chúng ta muốn chuyển SUNDAY thành SATURDAY với số bước sửa tối thiểu. Đầu tiên chúng ta lấy i = 2 và j = 4 – hai xâu tiền tố tương ứng là SU và SATU (giả sử chỉ số xâu bắt đầu từ 1). Các ký tự bên phải được căn theo ba cách.

*Trường hợp 1:* Căn hai ký tự U và U. Chúng bằng nhau, không cần phải sửa. Chúng ta xét bài toán con với i = 1 and j = 3, E(i-1, j-1).

*Trường hợp 2:* Căn ký tự ngoài cùng bên phải của xâu thứ nhất với không ký tự nào ở xâu thứ hai. Chúng ta cần một Thao tác xóa D. Chúng ta có bài toán con với i = 1 và j = 4, E(i-1, j).

*Trường hợp 3:* Căn ký tự ngoài cùng bên phải của xâu thứ hai với không ký tự nào ở xâu thứ nhất. Chúng ta cần một Thao tác chèn I ở đây. Chúng ta có bài toán con với i = 2 và j = 3, E(i, j-1).

Kết hợp tất cả các bài toán con, chúng ta có

E(i, j) = min( [E(i-1, j) + D], [E(i, j-1) + I], [E(i-1, j-1) + R nếu hai ký tự i, j khác nhau] )

Khi cả hai xâu đều rỗng, số thao tác là 0. Khi một trong hai xâu rỗng, số thao tác chính là độ dài xâu kia, tức là : E(0, 0) = 0, E(i, 0) = i, E(0, j) = j

**Dynamic Programming Method:**

Chúng ta có thể lập bảng kết quả các bài toán con và tra kết quả khi cần thiết.

```
/* Dynamic Programming implementation of edit distance */
#include<stdio.h>
```

```
#include<stdlib.h>
#include<string.h>

/* Change these strings to test the program */
#define STRING_1 "SUNDAY"
#define STRING_2 "SATURDAY"

#define SENTINEL (-1)
#define EDIT_COST (1)

/* Helper to display table */
void display(int *base, int row, int col)
{
    int r, c;

    for(r = 0; r < row; r++)
    {
        for(c = 0; c < col; c++)
        {
            printf("\t%4d", (base + r * col)1);
        }

        printf("\n\n");
    }
}

/* Returns minimum among a, b, c */
int minimum(int a, int b, int c)
{
    int min = c;

    /* Comparisons propotional to height
       of decision tree, here we need minimum 2 comparisons.
    */
    if( a < b )
    {
        if( a < c )
        {
            min = a;
        }
    }
    else
    {
        if( b < c )
        {
            min = b;
        }
    }

    return min;
}

/* Strings of size m and n are passed.
   Construct the table for X[0...m, m+1], Y[0...n, n+1]. */
int edit_distance( char *X, char *Y )
```

```
{
    /* Cost of alignment */
    int cost = 0;
    /* Cell data */
    int currentCell;
    int leftCell, topCell, cornerCell;

    /* We need an (m+1) x (n+1) matrix */
    int m = strlen(X) + 1; /* m is one more than X string length */
    int n = strlen(Y) + 1; /* n is one more than Y string length */

    /* table[m][n] */
    int *table = (int *)malloc( (m) * (n) * sizeof(int) );

    /* Indices */
    int i, j;

    /* Initialize the table (for visualisation) */
    for (i = 0; i < m; i++)
    {
        for (j = 0; j < n; j++)
        {
            *(table + i * n + j) = SENTINEL;
        }
    }

    /* Set up base cases */

    /* table[0][0] = 0 */
    *table = 0;
    for (i = 1; i < m; i++)
    {
        /* table[i][0] = i */
        *(table + i * n) = i;
    }
    for (j = 1; j < n; j++)
    {
        /* table[0][j] = j */
        *(table + j) = j;
    }

    /* Algorithm visualisation stub */
    printf("After Base Cases\n");
    display(table, m, n);

    /* Build the table in bottom-up fashion */
    for (i = 1; i < m; i++)
    {
        for (j = 1; j < n; j++)
        {
            /* table[i-1][j] */
            leftCell = *( table + (i-1)*n + j );
            leftCell += EDIT_COST;

            /* table[i][j-1] */
```

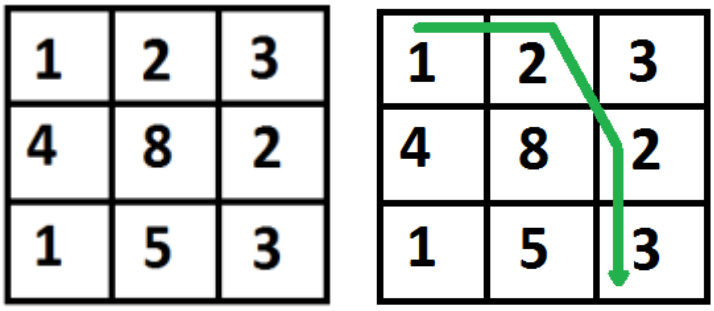
```
            topCell = *( table + i*n + (j-1) );
            topCell += EDIT_COST;

            /* table[i-1][j-1] */
            cornerCell = *( table + (i-1)*n + (j-1) );
            /* edit[(i-1), (j-1)] + 0 if X[i] == Y[j], 1 otherwise */
            cornerCell += (X[i-1] != Y[j-1]);

            /* Minimum cost of current cell */
            currentCell = minimum(leftCell, topCell, cornerCell);

            /* Fill in the next cell table[i][j] */
            *( table + (i)*n + (j) ) = currentCell;
        }

        /* Algorithm visualisation stub */
        printf("Table After Completion of Row\n");
        display(table, m, n);
        printf("\n\n");
    }

    /* Cost is in the last cell of table */
    cost = *(table + (m * n) - 1);

    free(table);

    return cost;
}

/* Reursive implementation */
int edit_distance_recursive( char *X, char *Y, int m, int n )
{
    /* Base cases */
    if(m == 0 && n == 0)
    {
        return 0;
    }

    if(m == 0)
    {
        return n;
    }

    if( n == 0 )
    {
        return m;
    }

    /* Recurse */
    int left   = edit_distance_recursive(X, Y, m-1, n) + 1;
    int right  = edit_distance_recursive(X, Y, m, n-1) + 1;
    int corner = edit_distance_recursive(X, Y, m-1, n-1) + (X[m-1] != Y[n-1]);

    return minimum(left, right, corner);
```

```
}

/* Driver stub to test edit_distance */
int main()
{
    char X[] = STRING_1;
    char Y[] = STRING_2;

    printf("Minimum edits required to convert %s into %s is %d\n",
                    X, Y, edit_distance( X, Y ) );
    printf("Minimum edits required to convert %s into %s is %d by
recursion\n",
                    X, Y, edit_distance_recursive( X, Y, strlen(X), strlen(Y)
) );

    return 0;
}
```

# ĐƯỜNG ĐI GIÁ TỐI THIỂU (Min Cost Path)

Cho ma trận giá cost[][] và vị trí (m, n) nằm trong cost[][], hãy viết một hàm trả về giá tối thiểu để đi từ ô (0,0) đến ô (m, n). Mỗi ô trong ma trận có 1 số nguyên dương - là giá khi đi qua ô đó. Tổng chi phí của một tuyến đường đến ô (m,n) là tổng giá của tất cả các ô nằm trên tuyến đường. Bạn chỉ có thể đi xuống dưới, sang phải hoặc dịch chéo xuống ô bên dưới. Tức là từ ô (i, j), có thể đến các ô sau: (i+1, j), (i, j+1) hoặc (i+1, j+1).

Ví dụ trong ma trận trên, giá đường đi tối thiểu tới ô (2, 2) là 8 ((0, 0) –> (0, 1) –> (1, 2) –> (2, 2))




Chúng ta để ý rằng đường đến ô (m, n) chắc chắn phải đi qua một trong 3 ô sau: (m-1, n-1) hoặc (m-1, n) hoặc (m, n-1). Do đó:

minCost(m, n) = min (minCost(m-1, n-1), minCost(m-1, n), minCost(m, n-1)) + cost[m][n]

Sau đây là chương trình đệ quy theo công thức nêu trên:

```
#include<stdio.h>
#include<limits.h>
#define R 3
#define C 3
```

```
int min(int x, int y, int z);

/* Returns cost of minimum cost path from (0,0) to (m, n) in mat[R][C]*/
int minCost(int cost[R][C], int m, int n)
{
   if (n < 0 || m < 0)
      return INT_MAX;
   else if (m == 0 && n == 0)
      return cost[m][n];
   else
      return cost[m][n] + min( minCost(cost, m-1, n-1),
                               minCost(cost, m-1, n),
                               minCost(cost, m, n-1) );
}

int min(int x, int y, int z)
{
   if (x < y)
      return (x < z)? x : z;
   else
      return (y < z)? y : z;
}

/* Driver program to test above functions */
int main()
{
   int cost[R][C] = { {1, 2, 3},
                      {4, 8, 2},
                      {1, 5, 3} };
   printf(" %d ", minCost(cost, 2, 2));
   return 0;
}

/* Dynamic Programming implementation of MCP problem */
#include<stdio.h>
#include<limits.h>
#define R 3
#define C 3

int min(int x, int y, int z);

int minCost(int cost[R][C], int m, int n)
{
     int i, j;

     // Instead of following line, we can use int tc[m+1][n+1] or
     // dynamically allocate memoery to save space. The following line is
     // used to keep te program simple and make it working on all compilers.
     int tc[R][C];

     tc[0][0] = cost[0][0];

     /* Initialize first column of total cost(tc) array */
     for (i = 1; i <= m; i++)
```

```
        tc[i][0] = tc[i-1][0] + cost[i][0];

     /* Initialize first row of tc array */
     for (j = 1; j <= n; j++)
        tc[0][j] = tc[0][j-1] + cost[0][j];

     /* Construct rest of the tc array */
     for (i = 1; i <= m; i++)
        for (j = 1; j <= n; j++)
            tc[i][j] = min(tc[i-1][j-1], tc[i-1][j], tc[i][j-1]) +
cost[i][j];

     return tc[m][n];
}

/* A utility function that returns minimum of 3 integers */
int min(int x, int y, int z)
{
   if (x < y)
      return (x < z)? x : z;
   else
      return (y < z)? y : z;
}

/* Driver program to test above functions */
int main()
{
   int cost[R][C] = { {1, 2, 3},
                      {4, 8, 2},
                      {1, 5, 3} };
   printf(" %d ", minCost(cost, 2, 2));
   return 0;
}
```

# BƯỚC NHẢY TỐI THIỂU ĐẾN ĐÍCH

Cho một mảng các số nguyên arr[], giá trị arr[i] là số bước nhảy tối đa có thể nhảy về phía trước tính từ vị trí i. Nếu đứng ở vị trí đầu tiên, sẽ mất bao nhiêu bước để nhảy đến vị trí cuối cùng. Nếu arr[i]=0, không được nhảy qua phần tử i.

Ví dụ:

```
Input: arr[] = {1, 3, 5, 8, 9, 2, 6, 7, 6, 8, 9}
Output: 3 (1-> 3 -> 8 ->9)
```

Phần tử đầu tiên là 1, do đó có thể nhảy đến 3. Phần tử thứ 2 là 3, do đó có thể nhảy tối đa ba bước, đến 5, 8 hoặc 9.

**Cách 1 (Đệ quy đơn giản)**
Giải pháp dễ nhất là nhảy từ vị trí thứ i và tiếp tục xét mọi vị trí có thể tới sau bước nhảy thứ nhất. Xác định số bước nhảy tối thiểu từ tất cả các vị trí thứ (i+1) này và lấy giá trị tối thiểu cộng 1 – đó chính là số bước nhảy tối thiểu từ vị trí thứ i. Kết quả bài toán là số bước nhảy tối thiểu từ vị trí 1.

```
minJumps(start, end) = Min ( minJumps(k, end) ) for all k reachable from
start
#include <stdio.h>
#include <limits.h>

// Tính số bước tối thiểu để nhảy từ vị trí arr[h] đến arr[l]
int  minJumps(int  arr[], int  l, int  h)
{
  // Base case: when source and destination are same
  if  (h == l)
    return  0;

  // When nothing is reachable from the given source
  if  (arr[l] == 0)
    return  INT_MAX;

  // Traverse through all the points reachable from arr[l]. Recursively
  // get the minimum number of jumps needed to reach arr[h] from these
  // reachable points.
  int  min = INT_MAX;
  for  (int  i = l+1; i <= h && i <= l + arr[l]; i++)
  {
     int  jumps = minJumps(arr, i, h);
     if(jumps != INT_MAX && jumps + 1 < min)
        min = jumps + 1;
  }

  return  min;
}

// Driver program to test above function
int  main()
{
 int  arr[] = {1, 3, 6, 3, 2, 3, 6, 8, 9, 5};
 int  n = sizeof(arr)/sizeof(arr[0]);
 printf("Minimum number of jumps to reach end is %d ", minJumps(arr, 0, n-1));
 return  0;
}
```

Chúng ta để ý có rất nhiều bài toán con trùng lặp, chẳng hạn minJumps(3, 9) sẽ được gọi hai lần vì arr[3] sẽ có thể là đích kế tiếp khi nhảy từ arr[1] và arr[2].

**Method 2 (Dynamic Programming)**
Trong cách này, chúng ta xây dựng mảng jumps[] từ trái sang phải sao cho jumps[i] là số bước tối thiểu để nhảy từ arr[0] đến arr[i]. Kết quả bài toán là jumps[n-1].

```
#include <stdio.h>
#include <limits.h>

// Returns minimum number of jumps to reach arr[n-1] from arr[0]
int minJumps(int arr[], int n)
{
   int *jumps = new int[n];  // jumps[n-1] will hold the result
   int i, j;

   if (n == 0 || arr[0] == 0)
      return INT_MAX;

   jumps[0] = 0;

   // Find the minimum number of jumps to reach arr[i]
   // from arr[0], and assign this value to jumps[i]
   for (i = 1; i < n; i++)
   {
      jumps[i] = INT_MAX;
      for (j = 0; j < i; j++)
      {
         if (i <= j + arr[j] && jumps[j] != INT_MAX)
         {
            jumps[i] = jumps[j] + 1;
            break;
         }
      }
   }
   return jumps[n-1];
}

// Driver program to test above function
int main()
{
   int arr[]= {1, 3, 6, 1, 0, 9};
   int size=sizeof(arr)/sizeof(int);
   printf("Minimum number of jumps to reach end is %d ", minJumps(arr,size));
   return 0;
}
```

Độ phức tạp: $O(n^2)$.

**Cách 3 (Dynamic Programming)**
Ở cách này, chúng ta xây dựng mảng jumps[] từ phải sang trái sao cho jumps[i] là số bước tối thiểu để nhảy từ arr[i] đến arr[n-1]. Kết quả của bài toán là arr[0].

```
int minJumps(int arr[], int n)
{
   int *jumps = new int[n];  // jumps[0] will hold the result
   int min;

   // Minimum number of jumps needed to reach last element
   // from last elements itself is always 0
   jumps[n-1] = 0;


       int i, j;

   // Start from the second element, move from right to left
   // and construct the jumps[] array where jumps[i] represents
   // minimum number of jumps needed to reach arr[m-1] from arr[i]
   for (i = n-2; i >=0; i--)
   {
      // If arr[i] is 0 then arr[n-1] can't be reached from here
      if (arr[i] == 0)
         jumps[i] = INT_MAX;

      // If we can direcly reach to the end point from here then
      // jumps[i] is 1
      else if (arr[i] >= n - i - 1)
         jumps[i] = 1;

      // Otherwise, to find out the minimum number of jumps needed
      // to reach arr[n-1], check all the points reachable from here
      // and jumps[] value for those points
      else
      {
         min = INT_MAX;  // initialize min value

         // following loop checks with all reachable points and
         // takes the minimum
         for (j = i+1; j < n && j <= arr[i] + i; j++)
         {
            if (min > jumps[j])
               min = jumps[j];
         }

         // Handle overflow
         if (min != INT_MAX)
```

```
            jumps[i] = min + 1;
          else
            jumps[i] = min; // or INT_MAX
        }
      }

    return jumps[0];
}
```

Độ phức tạp: $O(n^2)$.

# XÂU CON DÀI NHẤT KHÔNG CÓ KÝ TỰ TRÙNG

Cho một chuỗi, tìm độ lớn của xâu con lớn nhất không có ký tự trùng trong xâu con. Ví dụ xâu con dài nhất không có ký tự trùng của xâu "ABDEFGABEF" là xâu "BDEFGA" và xâu "DEFGAB" (đều có độ dài 6). Với xâu "BBBB", xâu con dài nhất không có ký tự trùng chỉ là "B" (độ dài 1).

**Cách 1**

**Ch**úng ta có thể sinh lần lượt từng xâu con, rồi kiểm tra xem trong mỗi xâu con đó có ký tự trùng không. Kiểm tra một xâu con có ký tự trùng không có độ phức tạp O(n), do đó độ phức tạp của thuật toán là $O(n^3)$.

**Cách 2 (Linear Time)**
Chúng ta sẽ sử dụng bảng để lưu lại chỉ số cuối cùng của ký tự đã thăm. Ý tưởng là xét dần từ trái sang phải, ở mỗi bước, xác định độ lớn của xâu con không trùng dài nhất max_len. Khi duyệt chuỗi, chúng ta xét xâu con không trùng hiện tại và giả sử xâu này có kích thước cur_len. Chúng ta dùng mảng visited[] là lưu lại chỉ số cuối cùng.  Với mỗi ký tự mới,

Let us talk about the linear time solution now. This solution uses extra space to store the last indexes of already visited characters. The idea is to scan the string from left to right, keep track of the maximum length Non-Repeating Character Substring (NRCS) seen so far. Let the maximum length be max_len. When we traverse the string, we also keep track of length of the current NRCS using cur_len variable. For every new character, we look for it in already processed part of the string (A temp array called visited[] is used for this purpose). If it is not present, then we increase the cur_len by 1. If present, then there are two cases:

**a)** The previous instance of character is not part of current NRCS (The NRCS which is under process). In this case, we need to simply increase cur_len by 1.
**b)** If the previous instance is part of the current NRCS, then our current NRCS changes. It becomes the substring staring from the next character of previous instance to currently scanned character. We also need to compare cur_len and max_len, before changing current NRCS (or changing cur_len).

## Implementation

```
#include<stdlib.h>
#include<stdio.h>
#define NO_OF_CHARS 256

int min(int a, int b);

int longestUniqueSubsttr(char *str)
{
    int n = strlen(str);
    int cur_len = 1;  // To store the lenght of current substring
    int max_len = 1;  // To store the result
    int prev_index;  // To store the previous index
    int i;
    int *visited = (int *)malloc(sizeof(int)*NO_OF_CHARS);

    /* Initialize the visited array as -1, -1 is used to indicate that
       character has not been visited yet. */
    for (i = 0; i < NO_OF_CHARS;  i++)
        visited[i] = -1;

    /* Mark first character as visited by storing the index of first
       character in visited array. */
    visited[str[0]] = 0;

    /* Start from the second character. First character is already processed
       (cur_len and max_len are initialized as 1, and visited[str[0]] is set
*/
    for (i = 1; i < n; i++)
    {
        prev_index =  visited[str[i]];

        /* If the currentt character is not present in the already processed
           substring or it is not part of the current NRCS, then do cur_len++
*/
        if (prev_index == -1 || i - cur_len > prev_index)
            cur_len++;

        /* If the current character is present in currently considered NRCS,
           then update NRCS to start from the next character of previous
instance. */
        else
        {
            /* Also, when we are changing the NRCS, we should also check
whether
              length of the previous NRCS was greater than max_len or not.*/
            if (cur_len > max_len)
                max_len = cur_len;

            cur_len = i - prev_index;
        }

        visited[str[i]] = i; // update the index of current character
```

```
    }

    // Compare the length of last NRCS with max_len and update max_len if
needed
    if (cur_len > max_len)
        max_len = cur_len;

    free(visited); // free memory allocated for visited

    return max_len;
}

/* A utility function to get the minimum of two integers */
int min(int a, int b)
{
    return (a>b)?b:a;
}

/* Driver program to test above function */
int main()
{
    char str[] = "ABDEFGABEF";
    printf("The input string is %s \n", str);
    int len =  longestUniqueSubsttr(str);
    printf("The length of the longest non-repeating character substring is
%d", len);

    getchar();
    return 0;
}
```

**Output**

```
  The input string is ABDEFGABEF
  The length of the longest non-repeating character substring is 6
```

**Time Complexity:** O(n + d) where n is length of the input string and d is number of characters in input string alphabet. For example, if string consists of lowercase English characters then value of d is 26.
**Auxiliary Space:** O(d)
**Algorithmic Paradigm:** Dynamic Programming

# 0-1 Knapsack Problem

Cho trước trọng lượng và giá trị của n đồ vật, hãy nhét một số đồ vật nào đó vào balô (tổng trọng lượn ba lô có thể mang được là W) sao cho tổng giá trị các đồ vật trong ba lô lớn nhất. Nói cách khác, có hai mảng số nguyên val[0..n-1] và wt[0..n-1] which represent values and wights associated with n items respectively. Also given an integer W which represents knapsack capacity, find out the maximum value subset of val[] such that sum of the weights of this subset

is smaller than or equal to W. You cannot break an item, either pick the complete item, or don't pick it (0-1 property).

A simple solution is to consider all subsets of items and calculate the total weight and value of all subsets. Consider the only subsets whose total weight is smaller than W. From all such subsets, pick the maximum value subset.

**1) Optimal Substructure:**
To consider all subsets of items, there can be two cases for every item: (1) the item is included in the optimal subset, (2) not included in the optimal set.
Therefore, the maximum value that can be obtained from n items is max of following two values.
1) Maximum value obtained by n-1 items and W weight (excluding nth item).
2) Value of nth item plus maximum value obtained by n-1 items and W minus wight of the nth item (including nth item).

If weight of nth item is greater than W, then the nth item cannot be included and case 1 is the only possibility.

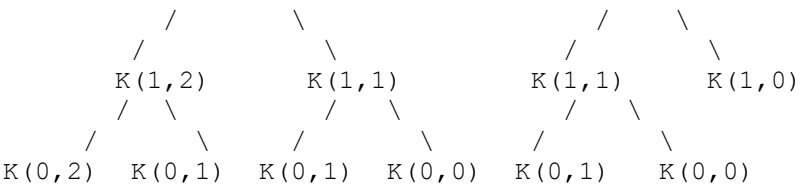
**2) Overlapping Subproblems**
Following is recursive implementation that simply follows the recursive structure mentioned above.

```
/* A Naive recursive implementation of 0-1 Knapsack problem */

#include<stdio.h>



// A utility function that returns maximum of two integers

int max(int a, int b) { return (a > b)? a : b; }



// Returns the maximum value that can be put in a knapsack of capacity W

int knapSack(int W, int wt[], int val[], int n)

{

   // Base Case

   if (n == 0 || W == 0)

       return 0;



   // If wight of the nth item is more than Knapsack capacity W, then
```

```
   // this item cannot be included in the optimal solution
   if (wt[n-1] > W)
       return knapSack(W, wt, val, n-1);

   // Return the maximum of two cases: (1) nth item included (2) not included
   else return max( val[n-1] + knapSack(W-wt[n-1], wt, val, n-1),
                    knapSack(W, wt, val, n-1)
                  );
}

// Driver program to test above function
int main()
{
    int val[] = {60, 100, 120};
    int wt[] = {10, 20, 30};
    int W = 50;
    int n = sizeof(val)/sizeof(val[0]);
    printf("%d", knapSack(W, wt, val, n));
    return 0;
}
```

It should be noted that the above function computes the same subproblems again and again. See the following recursion tree, K(1, 1) is being evaluated twice. Time complexity of this naive recursive solution is exponential (2^n).

```
In the following recursion tree, K() refers to knapSack().  The two
parameters indicated in the following recursion tree are n and W.
The recursion tree is for following sample inputs.
wt[] = {1, 1, 1}, W = 2, val[] = {10, 20, 30}

                        K(3, 2)         ---------> K(n, W)
                    /             \
                  /                 \
            K(2,2)                    K(2,1)
```



Recursion tree for Knapsack capacity 2 units and 3 items of 1 unit weight.

Since suproblems are evaluated again, this problem has Overlapping Subprolems property. So the 0-1 Knapsack problem has both properties (see this and this) of a dynamic programming problem. Like other typical Dynamic Programming(DP) problems, recomputations of same subproblems can be avoided by constructing a temporary array K[][] in bottom up manner. Following is Dynamic Programming based implementation.

```
// A Dynamic Programming based solution for 0-1 Knapsack problem

#include<stdio.h>


// A utility function that returns maximum of two integers

int max(int a, int b) { return (a > b)? a : b; }


// Returns the maximum value that can be put in a knapsack of capacity W

int knapSack(int W, int wt[], int val[], int n)

{

   int i, w;

   int K[n+1][W+1];


   // Build table K[][] in bottom up manner

   for (i = 0; i <= n; i++)

   {

       for (w = 0; w <= W; w++)

       {

           if (i==0 || w==0)

               K[i][w] = 0;

           else if (wt[i-1] <= w)
```

```
                K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1]],  K[i-1][w]);
            else
                K[i][w] = K[i-1][w];
        }
    }

    return K[n][W];
}

int main()
{
     int val[] = {60, 100, 120};
     int wt[] = {10, 20, 30};
     int W = 50;
     int n = sizeof(val)/sizeof(val[0]);
     printf("%d", knapSack(W, wt, val, n));
     return 0;
}
```

Time Complexity: O(nW) where n is the number of items and W is the capacity of knapsack.