Procesamiento de cadenas de caracteres

November 7, 2019

1 Procesamiento de cadenas de carácteres: lo básico

Elementos que dominar y practicar:

- Determinar el número de consonantes/vocales en una cadena de caracteres?
- Convertir en minúsculas, mayúsculas.
- Test de caracter numerico/alfabetico...

Dominar las funciones utilitarias de `ctype`:

```
#include <ctype.h>
int isalnum( int c);
int isalpha( int c);
int isdigit( int c);
int islower( int c);
int isupper( int c);
int tolower( int c);
int toupper( int c);
```

Dominar los **métodos** de la clase `string`:

```
size_t find (const string& str, size_t pos = 0) const;
int compare (size_t pos, size_t len, const string& str) const;
void push_back (char c);
string& insert (size_t pos, const string& str);
string substr (size_t pos = 0, size_t len = npos) const;
```

- Aprender una técnica de **tokenization** para separar las palabras de una linea por ejemplo.
- Manejar **histogramas** de palabras/sub-cadenas (bolsa de palabras).
- Usar **map** para las asociaciones palabra/frecuencia.

```
#include <string>
#include <vector>
#include <iostream>
#include <iterator>
#include <sstream>

using namespace std;
```

```
string s = "Les sanglots long des violons de l'automne";
stringstream ss(s);
istream_iterator<string> it(ss);
istream_iterator<string> end;

vector<string> tokens(it, end);
for (int i=0;i<tokens.size();i++) {
    cout << tokens[i] << endl;
}
```

UVA 902

Being able to send encoded messages during World War II was very important to the Allies. The messages were always sent after being encoded with a known password. Having a fixed password was of course insecure, thus there was a need to change it frequently. However, a mechanism was necessary to send the new password. One of the mathematicians working in the cryptographic team had a clever idea that was to send the password hidden within the message itself. The interesting point was that the receiver of the message only had to know the size of the password and then search for the password within the received text. A password with size N can be found by searching the text for the most frequent substring with N characters. After finding the password, all the substrings that coincide with the password are removed from the encoded text. Now, the password can be used to decode the message.

Your mission has been simplified as you are only requested to write a program that, given the size of the password and the encoded message, determines the password following the strategy given above. To illustrate your task, consider the following example in which the password size is three (N=3) and the text message is just "baababacb". The password would then be aba because this is the substring with size 3 that appears most often in the whole text (it appears twice) while the other six different substrings appear only once (baa;aab;bab;bac;acb).

Input: The input file contains several test cases, each of them consists of one line with the size of the password,$0 < N \leq 10$, followed by the text representing the encoded message. To simplify things, you can assume that the text only includes lower case letters.

Output: For each test case, your program should print as output a line with the password string.

```
#include <iostream>
#include <map>

using namespace std;

int main() {
    ios::sync_with_stdio(false);
    int nPattern;
    while (cin >> nPattern) {
        string s;
        map<string,int> counter;
        cin >> s;
```

```
        int vmax=0,imax;
        int l = s.size()-nPattern+1;
        for (int i=0;i<l;i++) {
            int v = counter[s.substr(i,nPattern)]++;
            if (v>vmax) {
                vmax = v; imax = i;
            }
        }
        cout << s.substr(imax,nPattern) << endl;
    }
    return 0;
}
```

Un tipo de problema técnico y clásico: el **parsing** de cadenas de caracteres, para satisfacibilidad: comprobar que una expresión satisface una cierta gramática con regla de producción de expresiones a partir de expresiones-base.

Suele requerir un enfoque recursivo.

UVA 622

Using the following grammar:

```
<expression> := <component> | <component> + <expression>

<component> := <factor> | <factor> * <component>

<factor> := <positive integer> | (<expression>)
```

write a program which analyses expressions conforming to the rules of this grammar and evaluates them, if the analysis has been successfully completed. It may be assumed that there is no overflow of float(C)/real(Pascal) numbers range.

Input: A integer n stating the number of expressions, then consecutive n lines, each containing an expression given as a character string.

Output: For each line value of the expression or output message ERROR if the expression does not follow the grammar.

```
#include <iostream>
#include <stack>

using namespace std;
int evaluateExpression(const string &s);

int evaluateFactor(const string &s) {
    if ((s[0]=='(') && (s[s.size()-1]==')')) {
        string ss = s.substr(1,s.size()-2);
        return evaluateExpression(ss);
```

```
    }
    // Should be a number
    int sum = 0;
    int p = 1;
    for (int i=0;i<s.size();i++) {
        if (!isdigit(s[s.size()-1-i]))
            return -1;
        sum+= p*(s[s.size()-1-i]-'0');
        p*=10;
    }
    return sum;
}

int evaluateComponent(const string &s) {
    if (s.size()==0) return -1;
    stack<int> st;
    // Search for *
    for (int i=0;i<s.size();i++) {
        if (s[i]=='(') st.push(1);
        else if (s[i]==')') st.pop();
        else if (st.size()==0 && s[i]=='*') {
            string s1 = s.substr(0 ,i);
            int r1 = evaluateFactor(s1);
            if (r1<0) return -1;
            string s2 = s.substr(i+1,s.size()-i+1);
            int r2 = evaluateComponent(s2);
            if (r2<0) return -1;
            return r1*r2;
        }
    }
    return evaluateFactor(s);
};

int evaluateExpression(const string &s){
    if (s.size()==0) return -1;
    stack<int> st;
    // Search for +
    for (int i=0;i<s.size();i++) {
        if (s[i]=='(') st.push(1);
        else if (s[i]==')') st.pop();
        else if (st.size()==0 && s[i]=='+') {
            string s1 = s.substr(0 ,i);
            int r1 = evaluateComponent(s1);
            if (r1<0) return -1;
            string s2 = s.substr(i+1,s.size()-i+1);
            int r2 = evaluateExpression(s2);
            if (r2<0) return -1;
            return r1+r2;
```

```
        }
    }
    // Evaluate a single component
    return evaluateComponent(s);
}

int main() {
    int nCases;
    string s,ss;
    cin >> nCases;
    getline(cin,s);
    for (int i=0;i<nCases;i++) {
        ss.clear();
        getline(cin,s);
        //cout << s << endl;
        bool valid = true;
        for (int j=0;j<s.size();j++)
            if (s[j]!=')' && s[j]!='(' && s[j]!='+' && s[j]!='*' && !isdigit(s[j])) {
                if (1) {
                    cout << "ERROR" << endl;
                    valid = false;
                    break;
                }
            } else {
                ss.push_back(s[j]);
            }
        if (valid) {
            int r = evaluateExpression(ss);
            if (r<0) cout << "ERROR" << endl;
            else cout << r << endl;
        }
    }
}
```

2 Problemas de matcheo de cadenas de caracteres

Problema de **buscar**, dentro de una cadena de caracteres de tamaño n, una **sub-cadena** dada de tamaño m.

- Se resuelve facilmente con la técnica más ingenua: recorrer index por index la cadena grande, y en paralelo recorrer index por index el patrón.
- En promedio $O(n)$.
- Considerar casos como:

  Cadena: XYZBLABLADEBLABLADEBLABLO

  Patrón: BLABLADEBLABLO

Para este tipo de casos, **complejidad hasta O(nm)**

Mejora posible? Pierdo la información de que las 13 primeras letras del patrón coincidieron... En particular "BLABL", que está presente anteriormente en el patrón.

2.1 Algoritmo de Knuth-Morris-Pratt

- Idea: en un mismatch donde la parte previamente matcheada no tiene nada en común (no hay repeticiones), seguir como en el algoritmo ingenuo.
- Idea: memorizar en el patrón **la estructura de repeticiones** para que, en caso de mismatch, se pueda **reusar comparaciones positivas hechas**, que pueden servir para otro test.
- En el ejemplo: Permitiría re-empezar las comparaciones al nivel del "A" mismatch, tomando por hechas las comparaciones positivas de "BLABL"

  Cadena: XYZBLABLADEBLABLADEBLABLO

  Patrón: BLABLADEBLABLO

Implica un **preprocesamiento de la cadena-patrón para guardar esta información**.

Recorrer la cadena patrón y notar, para cada letra en el cual potencialmente tendremos mismatch, la **posición de la mayor sub-cadena en común con el principio del patrón**.

Lo que requerimos es seguir la búsqueda **después del mayor prefijo igual a un sufijo terminando en la posición anterior a la del mismatch**.

Estructura/simetria fuerte en la cadena-patrón.

```
// Count the number of occurences of pattern within s
int kmpMatch(const string &s, const string &pattern) {
    int i=0,j=0,nc=0;
    while (i<s.size()) {
        while (j>=0 && s[i]!=pattern[j]) j = resets[j];
        i++; j++;
        if (j==pattern.size()) {
            nc++;
            j = resets[j];
        }
    }
    return nc;
}
```

- resets[j] contiene la posición siguiente en la mayor repetición como prefijo del sufijo matcheado **antes de un mismatch**.
- Podemos intentar comparar con la posición actual en *s*, sino, repetimos el proceso.

```
int resets[81];
void kmpPreprocess(const string &p) {
    int i=0,j=-1;
    resets[0]=-1;
    while (i<p.size()) { // For all caracters in the pattern
        while (j>=0 && p[i]!=p[j]) j=resets[j];
        i++;j++;
```

```
        resets[i]=j;
    }
}
```

BLABLADEBLABLO
-1 0 0 0 1 2 3 0 0 1 2 3 4 5

UVA 10056

A character string is said to have period k if it can be formed by concatenating one or more repetitions of another string of length k. For example, the string "abcabcabcabc" has period 3, since it is formed by 4 repetitions of the string "abc". It also has periods 6 (two repetitions of "abcabc") and 12 (one repetition of "abcabcabcabc"). Write a program to read a character string and determine its smallest period.

Input: The first line of the input file will contain a single integer N indicating how many test case that your program will test followed by a blank line. Each test case will contain a single character string of up to 80 non-blank characters. Two consecutive input will separated by a blank line.

Output: An integer denoting the smallest period of the input string for each input. Two consecutive output are separated by a blank line.

Idea: buscar las **occurencias de la cadena-patrón s dentro de s+s**. Si la cadena es periodica, entonces en cada periodo nos tiene que dar una occurencia. Contemos las occurencias encontradas.

Ese número va a ser igual a s.size()/p+1.

```
ios::sync_with_stdio(false);
int nCases;
cin >> nCases;
string s;
getline(cin,s);
for (int i=0;i<nCases;i++) {
    getline(cin,s);
    getline(cin,s);
    string s2 = s+s;
    kmpPreprocess(s);
    int nc = kmpMatch(s2,s);
    cout << s.size()/(nc-1) << endl;
    if (i<nCases-1)
        cout << endl;
}
```

UVA 12467

Alicia and Roberto like to play games. Today, Roberto is trying to guess a secret word that Alicia chose. Alicia wrote a long string S in a piece of paper and gave Roberto the following clues:

- The secret word is a non-empty substring (A substring of S is defined as any consecutive sequence of characters belonging to S. For example, if S = abcd then a, abcd, ab, bc and bcd are some of the substrings of S but ac, aa, aabbccdd and dcba are not substrings of S) of S (possibly equal to S).
- S starts with the secret word reversed.

Roberto knows Alicia very well, and he's sure that if there are several possible secret words that satisfy the clues above, Alicia must have chosen the longest one. Can you help him guess the secret word?

Input: The first line of the input file contains a single integer number T $\leq$ 150, the number of test cases. T lines follow, each with a single string S. S will only contain lowercase English letters. The length of S will not exceed one million characters.

Output: For each test case, output the secret word in one line.

La llave está en que la palabra viene **al inicio de S (al revés)**.

- Buscar occurencias parciales (a partir del inicio) es cosa facil con Kmp o cualquier algoritmo de matching (incluso el ingenuo).
- Revertimos S y buscamos el patrón S dentro de ella. Modificamos el Kmp para ir contando la mayor cadena al principio de S que encontremos en el reverso de S (corresponde al indice j).

```
#include <iostream>
#include <algorithm>
using namespace std;
int resets[1000001];
void kmpPreprocess(const string &p) {
    int i=0,j=-1;
    resets[0]=-1;
    while (i<p.size()) { // For all caracters in the pattern
        while (j>=0 && p[i]!=p[j]) j=resets[j];
        i++;j++;
        resets[i]=j;
    }
}
// Variant of kmpMatch to keep the largest partial match
int kmpMatch(const string &s, const string &pattern) {
    int i=0,j=0,jmax=-1;
    while (i<s.size()) {
        // Mismatch
        if (j>=0 && s[i]!=pattern[j]) {
            while (j>=0 && s[i]!=pattern[j]) j = resets[j];
        } else {
            if (j>jmax) {
                jmax= j;
            }
        }
```

```
        i++; j++;
    }
    return jmax;
}

int main() {
    ios::sync_with_stdio(false);
    int nCases;
    cin >> nCases;
    for (int i=0;i<nCases;i++) {
        string s;
        cin >> s;
        string s2(s); std::reverse(s2.begin(),s2.end());
        kmpPreprocess(s);
        int jmax= kmpMatch(s2,s);
        if (jmax>=s.size()) jmax = s.size()-1;
        for (int j=jmax;j>=0;j--)
            cout << s[j];
            cout << endl;
        }
    return 0;
}
```

Matching de cadenas en 2D (**sopa de letras**)

- Obviamente más complejo.
- Puede transformarse en un conjunto de 2h + 2w + 2 $*$ (h + w $-$ 1) búsquedas 1D de tipo KMP en el caso de que las palabras se pueden formar solo en direcciones horizontales/verticales/diagonales.

UVA 11283

Boggle is a classic word game played on a 4 by 4 grid of letters. The letter grid is randomly generated by shaking 16 cubes labeled with a distribution of letters similar to that found in English words. Players try to find words hidden within the grid. Words are formed from letters that adjoin horizontally, vertically, or diagonally. However, no letter may be used more than once within a single word. An example Boggle letter grid, showing the formation of the words "taxes" and "rise". The score awarded for a word depends on its length, with longer words being worth more points. Exact point values are shown in the table below. A word is only ever scored once, even if it appears multiple times in the grid.

In this problem, your task is to write a program that plays Boggle. Given a letter grid and a dictionary of words, you are to calculate the total score of all the words in the dictionary that can be found in the grid.

Input: The first line of the input file contains a number N, the number of Boggle games that follow. Each Boggle game begins with 16 capital letters arranged in a 4 by 4 grid, representing the board configuration for that game. A blank line always precedes the letter grid. Following the letter grid is a single number M ($1 \leq M \leq 100$), the number of words in your dictionary for that game. The next M lines contain the dictionary words, one per line, in no particular order. Each word consists of between 3 and 16 capital letters. No single word will appear in the dictionary more than once for a given Boggle game.

Output: For each Boggle game in the input, your program should output the total score for that game. Follow the format given in the sample output.

... o puede escribirse como problema recursivo de backtracking probando todas las configuraciones iniciales de la palabra.

```
#include <iostream>
#include <bitset>
#include <map>
#include <string.h>
#include <ctype.h>

using namespace std;
multimap<int,int> freqs;
char grid[4][4];
char dirs[8][2]={{0,1},{0,-1},{1,0},{-1,0},{1,1},{1,-1},{-1,1},{-1,-1}};

bool searchRecursive(const string &s,int index, int i,int j, bitset<16> b) {
    if (s[index]!=grid[i][j]) return false;
    if (index==s.size()-1) return true;
    // Set as occupied
    b.set(4*i+j);
    for (int d=0;d<8;d++) {
        int xf = i + dirs[d][0];
        int yf = j + dirs[d][1];
        if (xf>=0 && yf>=0 && xf<4 && yf<4 && !b[4*xf+yf]) {
            if (searchRecursive(s,index+1,xf,yf,b))
                return true;
        }
    }
    return false;
}

bool search(const string &s) {
    multimap<int,int>::iterator it = freqs.begin();
    for (;it!=freqs.end();it++) {
        int i = it->second/4;
        int j = it->second%4;
        bitset<16> bb;
        if (searchRecursive(s,0,i,j,bb))
            return true;
    }
    return false;
}

int main() {
    int nCases;
    cin >> nCases;
    string s;
```

```
getline(cin,s);
getline(cin,s);
for (int i=0;i<nCases;i++) {
    int nw;
    freqs.clear();
    int score=0;
    int hist[26];memset(&hist[0],0,sizeof(hist));
    for (int k=0;k<4;k++)
        for (int l=0;l<4;l++) {
            cin >> grid[k][l];
            grid[k][l]=tolower(grid[k][l]);
            hist[grid[k][l]-'a']++;
        }
    for (int k=0;k<4;k++)
        for (int l=0;l<4;l++) {
            freqs.insert(make_pair(hist[grid[k][l]-'a'],4*k+l));
        }
    cin >> nw;
    getline(cin,s);
    for (int k=0;k<nw;k++){
        getline(cin,s);
        string process;
        int histp[26];memset(&histp[0],0,sizeof(histp));
        for (int l=0;l<s.size() && isalpha(s[l]);l++) {
            process.push_back(tolower(s[l]));
            histp[tolower(s[l])-'a']++;
        }
        bool factible=true;
        for (int k=0;k<26;k++)
            if (hist[k]<histp[k]) {
                factible=false;
                break;
            }
        if(factible && search(process)) {
            if (process.size()>=3 && process.size()<=4)
                score++;
            else if (process.size()==5)
                score+=2;
            else if (process.size()==6)
                score+=3;
            else if (process.size()==7)
                score+=5;
            else if (process.size()>=8)
                score+=11;
        }
    }
    cout << "Score for Boggle game #" << i+1 << ": " << score << endl;
    if (i<nCases-1)
```

```
            getline(cin,s);
    }
    return 0;
}
```

3 Programación Dinámica con cadenas de caracteres

Varios problemas involviendo cadenas pueden resolverse usando **programación dinámica**:

- **Matching óptimo de cadenas o alineamiento** (con aplicaciones a bioinformática, visión...).
- Mayor **subsecuencia común**.
- Mayor **palindromo**.

Alineamiento (o calculo de la distancia de edición): para formular la programación dinámica, considerar 4 **acciones canonícas posibles entre las terminaciones parciales de cadenas A y B**, respectivamente en i y j para llevar A hacia B:

- copiar $A[i]$ en $B[j]$ si esos caracteres son iguales (mejor ganancia, e.g., $e > 0$),
- copiar $A[i]$ en $B[j]$ si no son iguales (ganancia $d < 0$),
- insertar un espacio en A (ganancia $h_A < 0$), i.e. saltarse $B[j]$,
- insertar un espacio en B (ganancia $h_B < 0$), i.e. saltarse $A[i]$.

Casos base:

- $C[0][0] = 0$.
- $C[0][j] = j * h_A$ (j inserciones de espacios en A).
- $C[i][0] = i * h_B$ (i inserciones de espacios en B).

Recursión:

$$C[i][j] = \max \begin{cases} C[i-1][j-1] + e \text{ si } A[i] = B[j] \text{ (matching)} \\ C[i-1][j-1] + d \text{ si } A[i] \neq B[j] \text{ (matching con ganancia muy baja)} \\ C[i-1][j] + h_B \text{ (inserción de 1 espacio en B)} \\ C[i][j-1] + h_A \text{ (inserción de 1 espacio en A)} \end{cases}$$

Algoritmo de Needleman-Wunsch.
$O(s_A s_B)$ tanto en tiempo como en memoria.
Ejemplo:

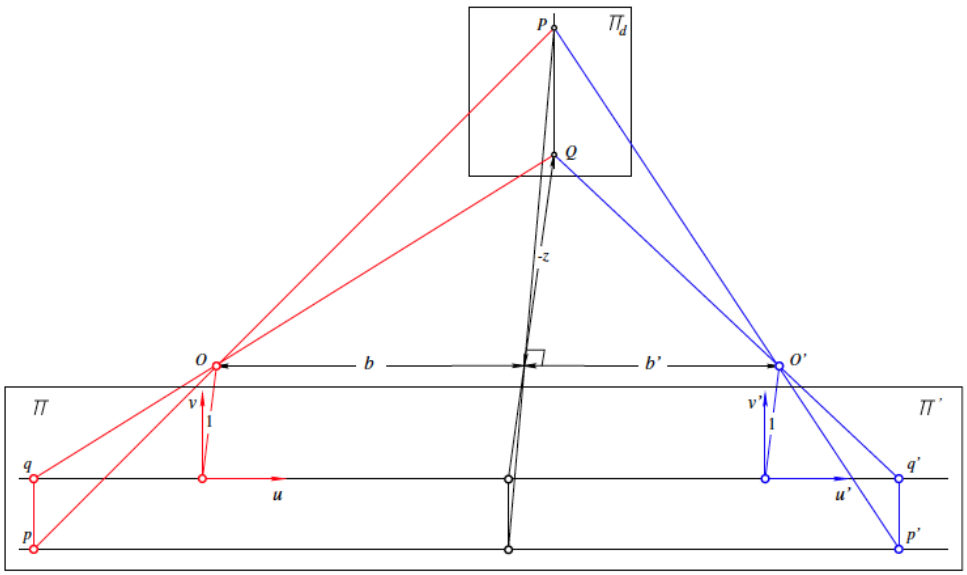
- Cadena A = ACAATCC
- Cadena B = AGCATGC

con $e = 2$, $d = h_A = h_B = -1$.
Estructura en la tabla:

- Emparejamientos (con igualdad o no): **diagonales**.
- Verticales: inserciones de espacio en B.
- Horizontales: inserciones de espacio en A.

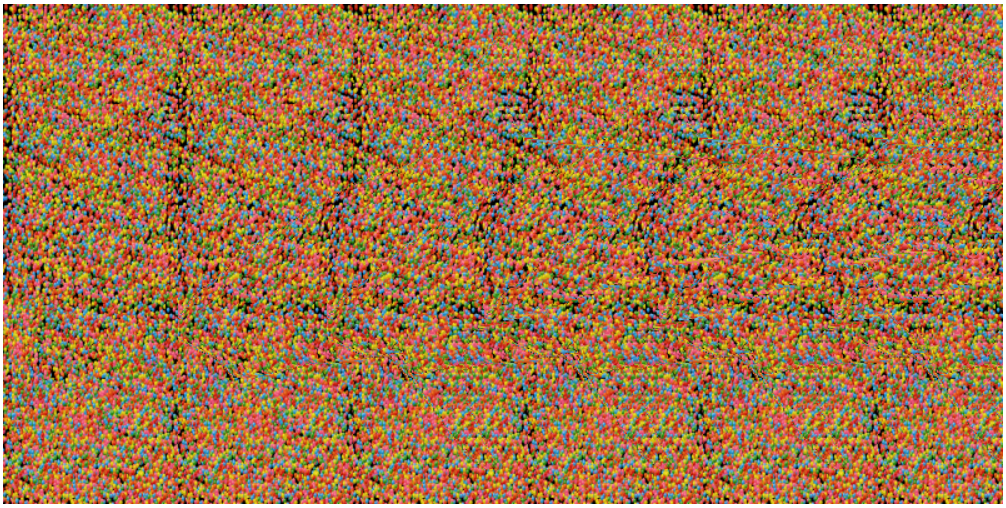
Mejoras:

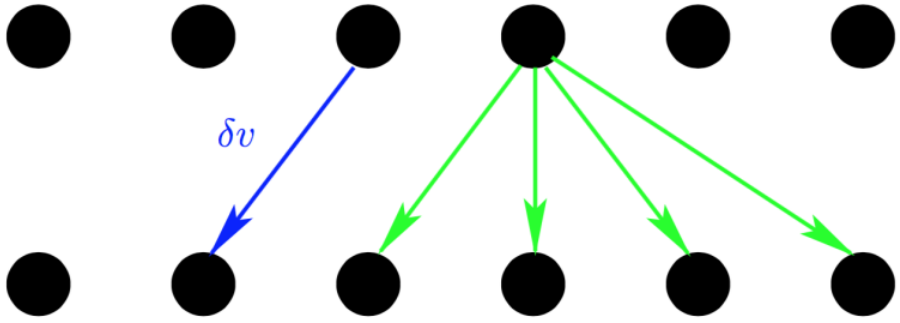
- **Reconstrucción** de la secuencia de acciones posible.
- Una versión para usar $O(min(s_A, s_B)$ en memoria?
- Una versión con máximo d inserciones de espacio en A o B?

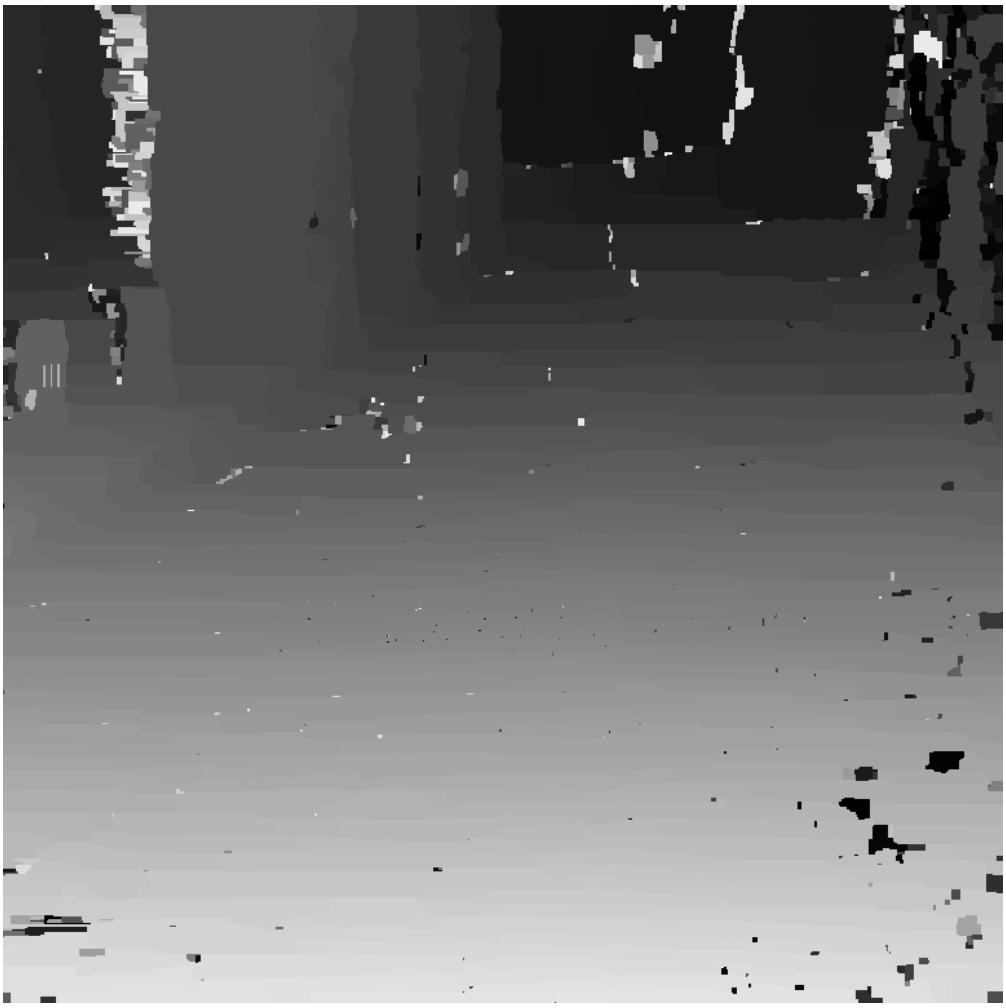


3.1 Aplicación a visión por computadora

- Problema muy similar.
- El matching local se hace en términos de similaridad de parches.




δv



3.2 Smith-Waterman

Algoritmo muy similar al de Needle-Wunch pero que detecta las **mejores secuencias emparajadas localmente**.

- Inicialización diferente (no costo/ganancia en descartar la primera parte de una de las cadenas):
$$C[0][0] = C[0][j] = C[i][0] = 0.$$
- Recursión modificada con la posibilidad de "resetear el matching":
$$C[i][j] = \max \begin{cases} C[i-1][j] + h_B \\ C[i][j-1] + h_A \\ C[i-1][j-1] + e \text{ si } A[i] = B[j] \\ C[i-1][j-1] + d \text{ si } A[i] \neq B[j] \\ 0 \end{cases}$$

  Complejidad **cuadrática**.

3.3 Penalización diferenciada de los gaps

$$C[i][j] = \max \begin{cases} \max_k C[i-k][j] + p(k) \\ \max_k C[i][j-k] + p(k) \\ C[i-1][j-1] + e \text{ si } A[i] = B[j] \\ C[i-1][j-1] + d \text{ si } A[i] \neq B[j] \end{cases}$$

- Permite **diferenciar los costos en función de la longitud de los gaps**!
- Con Needle-Wunsch: $p(k) \propto k$.
- En biología: $p(k)$ es tal que los gaps más largos tienen mayor ganancia!
- Complejidad en $O(n^3)$.
- En el caso particular de $p(k) = uk + v$ (modelo de penalidad afín), hay un algoritmo **cuadrático**.

3.4 Algoritm de Gotoh

Mantiene **3 matrices de ganancias** óptimas:

- $C[i][j]$, que ya conocemos
- $S[i][j]$, que contiene el costo óptimo de un alineamiento terminando con una inserción en A.
- $T[i][j]$, que contiene el costo óptimo de un alineamiento terminando con una inserción en B.

Para cada matriz, una actualizacion recursiva simple:

$$C[i][j] = \max \begin{cases} C[i-1][j-1] + e \text{ si } A[i] = B[j] \\ C[i-1][j-1] + d \text{ si } A[i] \neq B[j] \\ S[i][j] \\ T[i][j] \end{cases}$$

$$S[i][j] = \max \begin{cases} S[i][j-1] + u \\ C[i][j-1] + u + v, \end{cases}$$

y

$$T[i][j] = \max \begin{cases} T[i-1][j] + u \\ C[i-1][j] + u + v. \end{cases}$$

UVA 526

String Distance is a non-negative integer that measures the distance between two strings. Here we give the definition. A transform list is a list of strings, where each string, except for the last one, can be changed to the string followed by adding a character, deleting a character or replacing a character. The length of a transform list is the count of strings minus 1 (that is the count of operations to transform these two strings). The distance between two strings is the length of a transform list from one string to the other with the minimal length. You are to write a program to calculate the distance between two strings and give the corresponding transform list.

Input: Input consists a sequence of string pairs, each string pair consists two lines, each string occupies one line. The length of each string will be no more than 80.

Output: For each string pair, you should give an integer to indicate the length between them at the first line, and give a sequence of command to transform string1 to string 2. Each command is a line lead by command count, then the command. A command must be

Insert pos,value

Delete pos

Replace pos,value

where pos is the position of the string and pos should be between 1 and the current length of the string (in Insert command, pos can be 1 greater than the length), and value is a character. Actually many command lists can satisfy the request, but only one of them is required. Print a blank line between consecutive datasets.

3.5 Mayor común sub-sequencia

Reducible al problema anterior de matcheo de secuencias:

- descartar la opción de integrar caracteres no-iguales ($d = -\infty$).
- penalizar las inserciones con $h_A = h_B = 0$.
- ganar con cada elemento en común (matches) $e = 1$.

Complejidad en O(nm).

UVA 10192

You are planning to take some rest and to go out on vacation, but you really don't know which cities you should visit. So, you ask your parents for help. Your mother says "My son, you MUST visit Paris, Madrid, Lisboa and London. But it's only fun in this order." Then your father says: "Son, if you're planning to travel, go first to Paris, then to Lisboa, then to London and then, at last, go to Madrid. I know what I'm talking about."

Now you're a bit confused, as you didn't expected this situation. You're afraid that you'll hurt your mother if you follow your father's suggestion. But you're also afraid to hurt your father if

you follow you mother's suggestion. But it can get worse, because you can hurt both of them if you simply ignore their suggestions! Thus, you decide that you'll try to follow their suggestions in the better way that you can. So, you realize that the "Paris-Lisboa-London" order is the one which better satisfies both your mother and your father. Afterwards you can say that you could not visit Madrid, even though you would've liked it very much. If your father have suggested the "London-Paris-Lisboa-Madrid" order, then you would have two orders, "Paris-Lisboa" and "Paris-Madrid", that would better satisfy both of your parent's suggestions. In this case, you could only visit 2 cities. You want to avoid problems like this one in the future. And what if their travel suggestions were bigger? Probably you would not find the better way very easy. So, you decided to write a program to help you in this task. You'll represent each city by one character, using uppercase letters, lowercase letters, digits and the space. Thus, you can have at most 63 different cities to visit. But it's possible that you'll visit some city more than once. If you represent Paris with 'a', Madrid with 'b', Lisboa with 'c' and London with 'd', then your mother's suggestion would be 'abcd' and you father's suggestion would be 'acdb' (or 'dacb', in thesecond example).

The program will read two travel sequences and it must answer how many cities you can travel to such that you'll satisfy both of your parents and it's maximum.

Input: The input will consist on an arbitrary number of city sequence pairs. The end of input occurs when the first sequence starts with an '#' character (without the quotes). Your program should not process this case. Each travel sequence will be on a line alone and will be formed by legal characters (as defined above). All travel sequences will appear in a single line and will have at most 100 cities.

Output: For each sequence pair, you must print the following message in a line alone: Case #d: you can visit at most K cities. Where d stands for the test case number (starting from 1) and K is the maximum number of cities you can visit such that you'll satisfy both you father's suggestion and you mother's suggestion.

```
#include <iostream>
using namespace std;
int values[102][102];
int main() {
    string s1,s2;
    int nCase=1;
    values[0][0]=0;
    for (int i=0;i<102;i++) {
        values[i][0]=0;
        values[0][i]=0;
    }
    while (getline(cin,s1) && s1[0]!='#') {
        getline(cin,s2);
    for (int i=0;i<s1.size();i++)
        for (int j=0;j<s2.size();j++) {
            int m = values[i][j+1];
            int n = values[i+1][j];
            values[i+1][j+1] = max(m,n);
            if (s1[i]==s2[j])
                values[i+1][j+1] = max(values[i+1][j+1],values[i][j]+1);
```

```
        }
        cout << "Case #" << nCase << ": you can visit at most " << values[s1.size()][s2.size()]
        nCase++;
    }
    return 0;
}
```

UVA 10635

In an nxn chessboard, Prince and Princess plays a game. The squares in the chessboard are numbered 1;2;3;. . . ;nxn, as shown below. Prince stands in square 1, make p jumps and finally reach square nxn. He enters a square at most once. So if we use x p to denote the p-th square he enters, then x 1 , x 2 , ..., x p+1 are all different. Note that x 1 = 1 and x p+1 = n $\times$ n. Princess does the similar thing { stands in square 1, make q jumps and finally reach square n $\times$ n. We use y 1 , y 2 , ..., y q+1 to denote the sequence, and all q + 1 numbers are different. Figure 2 belows show a 3 $\times$ 3 square, a possible route for Prince and a different route for Princess. The Prince moves along the sequence: 1 -> 7 -> 5 -> 4 -> 8 -> 3 -> 9 (Black arrows), while the Princess moves along this sequence: 1 -> 4 -> 3 -> 5 -> 6 -> 2 ->8 -> 9 (White arrow).

For example, if the Prince ignores his 2nd, 3rd, 6th jump, he'll follow the route: 1 -> 4 -> 8 ->9. If the Princess ignores her 3rd, 4th, 5th, 6th jump, she'll follow the same route: 1 -> 4 -> 8 -> 9, (The common route is shown in figure 3) thus satisfies the King, shown above. The King wants to know the longest route they can move together, could you tell him?

Input: The first line of the input contains a single integer t ($1 \leq t \leq 10$), the number of test cases followed. For each case, the first line contains three integers n, p, q ($2 \leq n \leq 250$, $1 \leq p$; $q < n \times n$). The second line contains p + 1 different integers in the range [1, n $\times$ n], the sequence of the Prince. The third line contains q + 1 different integers in the range [1, n $\times$ n], the sequence of the Princess.

Output: For each test case, print the case number and the length of longest route. Look at the output for sample input for details.

El LCS decrito arriba no funcionará! (cada dimensión es en 250x250).

- Aprovechar del hecho de que estas secuencias tienen la propriedad de no pasar dos veces por el mismo número.
- Enumerar los puntos pasados por el Prince.
- Asignarles esos indices a los (comunes) de la Princess.
- Encontrar una secuencia común corresponde a encontrar una **secuencia creciente** entre los indices de la Princess.
- Problema LIS, con programación dinámica.

```
#include <iostream>
#include <string.h>
#include <algorithm>
using namespace std;
int indicesRef[250*250+1];
int indices[250*250+1];
int lis[250*250+1];
int big=250*250+200;
int main() {
    ios::sync_with_stdio(false);
    int nCases;
```

```
    cin >> nCases;
    for (int i=0;i<nCases;i++) {
        // Init
        memset(&indicesRef[0],0,sizeof(indicesRef));
        memset(&indices[0],0,sizeof(indices));
        memset(&lis[0],0,sizeof(lis));
        int n,p,q;
        cin >> n >> p >> q;
        // Take first sequence as reference
        for (int j=1;j<=p+1;j++) {
            int d; cin >> d;
            indicesRef[d] = j;
        }
        // Get the second sequence in terms of the first
        int nOther=0;
        for (int j=1;j<=q+1;j++) {
            int d; cin >> d;
            if (indicesRef[d]) // If in prince's path
                indices[nOther++]=indicesRef[d]; // Keep the place in prince's path only
        }
        // Now find the LIS length for the second
        int lisL = -1;
        for (int j=1;j<=nOther;j++)
            lis[j]=big;
        for (int j=0;j<nOther;j++) {
            int pos = int(upper_bound(lis+0,lis+nOther,indices[j])-(lis+0));
            lis[pos]=indices[j];
            if (pos>lisL) lisL = pos;
        }
        cout << "Case " << i+1 << ": " << lisL << endl;
    }
    return 0;
}
```

3.6 Palindromos

Programación dinámica: Para una secuencia s de tamaño n

- Considerar los palindromos más largos dentro de la subsecuencia $i..j$: arreglo $C[i][j]$.
- Se inicializa con:

$$C[i][i] = 1$$
$$C[i][i+1] = 2 \text{ si } s[i] = s[i+1], 1 \text{ sino}$$

- Se completa con:

$$C[i][j] = 2 + C[i+1][j-1] \text{ si } s[i] = s[j],$$
$$C[i][j] = \max C[i+1][j], C[i][j-1] \text{ sino}$$

UVA 11258

John was absurdly busy for preparing a programming contest recently. He wanted to create a ridiculously easy problem for the contest. His problem was not only easy, but also boring: Given

a list of non-negative integers, what is the sum of them? However, he made a very typical mistake when he wrote a program to generate the input data for his problem. He forgot to print out spaces to separate the list of integers. John quickly realized his mistake after looking at the generated input file because each line is simply a string of digits instead of a list of integers.

He then got a better idea to make his problem a little more interesting: There are many ways to split a string of digits into a list of non-zero-leading (0 itself is allowed) 32-bit signed integers.

What is the maximum sum of the resultant integers if the string is split appropriately?

Input: The input begins with an integer N ($\leq$ 500) which indicates the number of test cases followed. Each of the following test cases consists of a string of at most 200 digits.

Output: For each input, print out required answer in a single line.

- Programación dinámica 1D: para cualquier sub-cadena que termina con la cadena principal, y empieza en i, podemos distinguir todos los diferentes papeles del primer dígito s[i]. Y determinar la mayor suma posible para esta subcadena C[i].
- s[i] puede ser un dígito solo, o el digito encabezando un numero de 2, 3, 4,. . . 9 digitos (mientras queda en el limite de los 32 bits). Con los demas digitos de la cadena, tenemos un problema de tamano reducido ya procesado.
- Caso de arranque: C[s-1], donde s es el tamano de la cadena.

```
#include <iostream>
#include <string.h>
using namespace std;
long long C[256];
string biggest="2147483647";
long long pow[10];
int values[256];

int main() {
    int nCases;
    ios::sync_with_stdio(false);
    cin >> nCases;
    // Precompute power of 10
    pow[0]=1;
    for (int i=1;i<10;i++)
        pow[i]=10*pow[i-1];
    for (int i=0;i<nCases;i++) {
        memset(&C[0],0,sizeof(C));
        string s;
        cin >> s;
        for (int j=0;j<s.size();j++)
            values[j]=s[j]-'0';
        C[s.size()-1] = values[s.size()-1];
        for (int j=s.size()-2;j>=0;j--) {
            int d = values[j];
            if (d==0)
                C[j]=C[j+1];
            else {
```

```
                // All options of forming a 32-bit number starting with this digit
                for (int k=0;k<min(biggest.size(),s.size()-j);k++) {
                    // Evaluate the corresponding number, led by d
                    // If superior to largest int, then abort
                    long long sum = d*pow[k];
                    for (int l=1;l<=k;l++)
                        sum += pow[k-l]*values[j+l];
                    if (sum>2147483647)
                        continue;
                    //
                    C[j]=max(C[j],sum+C[j+k+1]);
                }
            }
        }
        cout << C[0] << endl;
    }
    return 0;
}
```