Najdłuższy wspólny podciąg

#algorytm_LCS

Przy pomocy metody programowania dynamicznego można rozwiązać wiele problemów algorytmicznych, w szczególności ważny problem znajdowania najdłuższego wspólnego podciągu dwóch ciągów znaków (ang. **Longest Common Subsequence**, **LCS**, wymawiaj: *longest komon sabsekłens*).

Generalnie, *podciągiem* ciągu znaków s nazywa się ciąg znaków utworzony z wybranych znaków ciągu s z zachowaniem ich kolejności. Na przykład jeśli s to **ABCDA**, wtedy jego przykładowymi podciągami są **AB**, **BCA** czy **AA**. Ciąg znaków s jest także swoim własnym podciągiem, również pusty ciąg znaków jest podciągiem ciągu s.

Najdłuższy wspólny podciąg ciągów s oraz t to taki ciąg znaków w, który jest zarazem podciągiem s oraz podciągiem t i do tego ma największą długość. Na przykład najdłuższy wspólny podciąg ciągów **ABCDA** oraz **BADEA** to ciąg **BDA**.

Łatwo zauważyć, że taki najdłuższy podciąg może nie być określony jednoznacznie, na przykład dla ciągów **ABCDA** oraz **BCEAD** może to być zarówno **BCA**, jak i **BCD** – natomiast jego długość jest zawsze wyznaczona jednoznacznie (w tym przykładzie wynosi 3).

Długość najdłuższego wspólnego podciągu (LCS)

Załóżmy, że chcemy wyznaczyć długość LCS dla dwóch ciągów znaków s oraz t o długościach odpowiednio n oraz m:

$$s = s_1 s_2 \cdots s_{n-1} s_n \equiv s(1, n),$$
$$t = t_1 t_2 \cdots t_{m-1} t_m \equiv t(1, m).$$

Oznaczmy poszukiwaną długość przez d – na początku równa jest 0.

Porównujemy końcowe znaki obydwu ciągów, czyli s_n oraz t_m. Jeśli są równe, to zaliczamy jeden z nich (są równe, więc obojętnie który) do LCS, zaś d otrzymuje wartość $1 +$ długość LCS dla ciągów $s(1, n-1)$ oraz $t(1, m-1)$ – czyli dla ciągów z usuniętymi końcowymi znakami.

Jeśli natomiast $s_n \neq t_m$, wtedy d jest równa większej z poniższych wartości:

- długość LCS dla ciągów $s(1, n)$ oraz $t(1, m-1)$,
- długość LCS dla ciągów $s(1, n-1)$ oraz $t(1, m)$.

Powyższy algorytm jest w oczywisty sposób rekurencyjny: opisane kroki trzeba powtarzać, aż skończy nam się któryś z ciągów znaków (lub oba naraz).

W celu efektywnej relizacji tego algorytmu wykorzystamy poznaną już metodę wstępującą, która posłuży się pomocniczą tablicą (nazwiemy ją $P[\,][\,]$), gdzie będziemy *spamiętywać* cząstkowe rezultaty. Jak zapewne Czytelnik pamięta, metoda ta nie wymaga użycia rekurencji jako takiej. Nowością jednak będzie fakt, że pomocnicza tablica $P[\,][\,]$ będzie tablicą dwuwymiarową: jej wiersze będą odpowiadać kolejnym znakom pierwszego ciągu, a kolumny – kolejnym znakom drugiej ciągu.

Dla przykładowych ciągów $s = \mathbf{BBCC}$ oraz $t = \mathbf{ABCBAC}$ mamy następującą tablicę:

	$t \rightarrow$	**A**	**B**	**C**	**B**	**A**	**C**
$s \downarrow$	0	0	0	0	0	0	0
B	0						
B	0						
C	0						
C	0						

Jak widać, tablica $P[\,][\,]$ posiada dodatkowy wiersz o numerze 0 oraz dodatkową kolumnę również o numerze 0 – ma to na celu uproszczenie zapisu algorytmu. Te dodatkowe elementy tablicy zostały wstępnie wypełnione zerami. W pozostałych komórkach będziemy umieszczać kolejne długości LCS wyliczone dla początkowych fragmentów ciągów s i t. Fragmenty te będą stopniowo zwiększane, aż dojdziemy do prawego dolnego rogu tablicy, w którym znajdzie się długość LCS dla pełnych ciągów s oraz t.

Zaczniemy po kolei wypełniać komórki tablicy $P[\,][\,]$. Komórkę, w której w danym kroku będziemy wpisywać wartość, oznaczymy kolorem żółtym:

	$t \rightarrow$	**A**	**B**	**C**	**B**	**A**	**C**
$s \downarrow$	0	0	0	0	0	0	0
B	0						
B	0						
C	0						
C	0						

Dla tej komórki (oznaczmy ją $P[i][j]$) patrzymy na odpowiadające jej znaki:

- w tym samym wierszu (i) w pierwszej kolumnie (znak z ciągu s),
- w tej samej kolumnie (j) w pierwszym wierszu (znak z ciągu t).

Jeśli znaki te byłyby równe, wpisalibyśmy wartość $P[i-1][j-1]+1$, gdyż uzyskalibyśmy nowy znak potencjalnie należący do LCS. (Wiersze numerowane są od góry do dołu, a kolumny od lewej do prawej.)

Jeśli znaki te nie są równe (a taką mamy tu sytuację, gdyż $\mathbf{B} \neq \mathbf{A}$), wpisujemy większą z wartości $P[i-1][j]$ oraz $P[i][j-1]$. Umówmy się, że jeśli te wartości są równe, wtedy wybieramy wartość z lewej strony. Niby to samo, ale jednak nie.

Oto rezultat:

	$t \rightarrow$	**A**	**B**	**C**	**B**	**A**	**C**
$s \downarrow$	0	0	0	0	0	0	0
B	0	0					
B	0						
C	0						
C	0						

Wpisujemy kolejną komórkę – znów 0:

	$t \rightarrow$	**A**	**B**	**C**	**B**	**A**	**C**
$s \downarrow$	0	0	0	0	0	0	0
B	0	0					
B	0	0					
C	0						
C	0						

I tak będzie aż do końca tej kolumny:

	$t \rightarrow$	**A**	**B**	**C**	**B**	**A**	**C**
$s \downarrow$	0	0	0	0	0	0	0
B	0	0					
B	0	0					
C	0	0					
C	0	0					

Zaczynamy kolejną kolumnę:

	$t \rightarrow$	**A**	**B**	**C**	**B**	**A**	**C**
$s \downarrow$	0	0	0	0	0	0	0
B	0	0					
B	0	0					
C	0	0					
C	0	0					

Tu mamy inną sytuację, gdyż znaki się zgadzają – tu **B** i tu **B** – zatem weźmiemy wartość z zielonej komórki (po skosie) powiększoną o 1:

	$t \rightarrow$	**A**	**B**	**C**	**B**	**A**	**C**
$s \downarrow$	0	0	0	0	0	0	0
B	0	0	1				
B	0	0					
C	0	0					
C	0	0					

Dokładnie to samo dzieje się w komórce poniżej:

	$t \rightarrow$	**A**	**B**	**C**	**B**	**A**	**C**
$s \downarrow$	0	0	0	0	0	0	0
B	0	0	1				
B	0	0	1				
C	0	0					
C	0	0					

Zaraz, zaraz, nie rozpędzajmy się. Wygląda na to, że w ten sposób znajdziemy długość LCS, ale co z samym podciągiem? Jak go wyznaczyć? Otóż okazuje się, że da się to bardzo prosto zrobić – w każdej komórce tablicy $P[][]$ trzeba umieścić dodatkową informację, w oparciu o którą sąsiednią komórkę była wyliczana jej zawartość.

Przyjmijmy takie oznaczenia:

- $\leftarrow$ – jeśli wykorzystano komórkę po lewej stronie,
- $\uparrow$ – jeśli wykorzystano komórkę powyżej,
- $\nwarrow$ – jeśli wykorzystano komórkę po skosie.

Zatem nasza tablica $P[][]$ na dotychczasowym etapie wypełniania przedstawia się tak:

	$t \rightarrow$	**A**	**B**	**C**	**B**	**A**	**C**
$s \downarrow$	0	0	0	0	0	0	0
B	0	$\leftarrow$ 0	$\nwarrow$ 1				
B	0	$\leftarrow$ 0	$\nwarrow$ 1				
C	0	$\leftarrow$ 0					
C	0	$\leftarrow$ 0					

Uzupełnimy teraz pozostałe komórki tablicy według przedstawionych wcześniej reguł:

	$t \rightarrow$	**A**	**B**	**C**	**B**	**A**	**C**
$s \downarrow$	0	0	0	0	0	0	0
B	0	$\leftarrow$ 0	$\nwarrow$ 1	$\leftarrow$ 1	$\nwarrow$ 1	$\leftarrow$ 1	$\leftarrow$ 1
B	0	$\leftarrow$ 0	$\nwarrow$ 1	$\leftarrow$ 1	$\nwarrow$ 2	$\leftarrow$ 2	$\leftarrow$ 2
C	0	$\leftarrow$ 0	$\leftarrow$ 1	$\nwarrow$ 2	$\leftarrow$ 2	$\leftarrow$ 2	$\nwarrow$ 3
C	0	$\leftarrow$ 0	$\nwarrow$ 1	$\nwarrow$ 2	$\leftarrow$ 2	$\leftarrow$ 2	$\nwarrow$ 3

Zatem długość LCS w tym przykładzie wynosi 3. W następnej sekcji przedstawimy sposób konstrukcji samego ciągu LCS, a teraz zajmiemy się implementacją funkcji `LCS_length()`.

Pierwsza techniczna uwaga dotyczy sposobu implementacji symboli niebieskich strzałek – najprościej zrobić to przy użyciu zdefiniowanych stałych liczbowych:

```
#define UP 0
#define LEFT 1
#define NW 2
```

Stała *UP* (wymawiaj: *ap*) oznacza strzałkę do góry, stała *LEFT* – strzałkę w lewo, stała *NW* (ang. **North-West**, wymawiaj: *norθ łest*, północny zachód) oznacza strzałkę ukośną.

Konkretne wartości tych stałych (tutaj odpowiednio: 0, 1, 2) nie mają żadnego znaczenia – muszą tylko być różne.

Ponieważ przechowywanie dwóch liczb w jednej komórce tablicy $P[][]$ jest z lekka uciążliwe,*, więc radzimy użycie dwóch oddzielnych tablic o wartościach całkowitych $P[][]$ oraz $Q[][]$ – będą to tablice dwuwymiarowe o dokładnie takich samych wymiarach. W tablicy $P[][]$ przechowamy długości cząstkowych ciągów, a w tablicy $Q[][]$ znajdą się stałe odpowiadajace strzałkom. Tak naprawdę, to nie użyjemy zwykłych tablic, tylko wektorów, których elementami będą wektory liczb całkowitych.

Oto nagłówek funkcji:†

```
int LCS_length(string s, string t, vector<vector<int>> &Q)
{
  . . .
}
```

Zmienna $P[][]$ będzie zmienną lokalną (nie jest potrzebna na zewnątrz funkcji), natomiast rzeczywistym rezultatem funkcji będzie długość LCS oraz zmienna $Q[][]$ (przekazywana przez referencję).

Zaczynamy od ustalenia wartości n oraz m i zadeklarowania zmiennej $P[][]$:

```
int n = s.size(), m = t.size();
vector<vector<int>> P;
```

Musimy popełnić małe oszustwo: jak pamiętamy, numeracja elementów `string`-u zaczyna się od indeksu 0 (dla przykładu od 0 do $n-1$ w ciągu s). My jednak potrzebujemy numeracji od 1 (od 1 do n dla ciągu s). W tym celu na początku ciągów s i t można dołożyć dodatkowy znak (dowolny), aby zwiększyć o 1 wartości indeksów pozostałych znaków:

```
s = '#' + s;
t = '#' + t;
```

Zarówno zmienna $P[][]$ jak i $Q[][]$ powinny otrzymać ustalony rozmiar: $n+1$ wierszy i $m+1$ kolumn. Musimy zrobić to na dwa tempa – tak robi się to dla wektorów wektorów (czyli struktur dwuwymiarowych).

Najpierw *alokujemy*‡ wektory zewnętrzne:

```
P.resize(n+1);
Q.resize(n+1);
```

*Oczywiście jest to możliwe przy użyciu choćby kontenera `pair`.

†W stylu C++11.

‡Czyli rezerwujemy dla nich miejsce w pamięci.

Następnie alokujemy kolejne elementy tych wektorów, czyli wiersze tablic $P[][]$ oraz $Q[][]$:

```
for(int i = 0; i <= n; i++)
{
  P[i].resize(m + 1);
  Q[i].resize(m + 1);
}
```

Teraz możemy przystąpić do wypełniania zawartości obu tablic (zmienna i numeruje wiersze, a zmienna j – kolumny):

```
for(int j = 1; j <= m; j++)
  for(int i = 1; i <= n; i++)
    if(s[i] == t[j])
    {
      P[i][j] = P[i - 1][j - 1] + 1;
      Q[i][j] = NW;
    }
    else
      . . .
```

Czyli jeśli odpowiednie znaki z obu ciągów są równe, wybieramy komórkę po skosie (`NW`) i dodajemy do niej 1. W przeciwnym razie kopiujemy wartość z lewej strony (`LEFT`) lub z góry (`UP`):

```
if(P[i - 1][j] >= P[i][j - 1])
{
  P[i][j] = P[i - 1][j];
  Q[i][j] = LEFT;
}
else
{
  P[i][j] = P[i][j - 1];
  Q[i][j] = UP;
}
```

Na koniec funkcja powinna zwrócić wartość z prawego dolnego rogu tablicy:

```
return P[n][m];
```

Drugim rezultatem tej funkcji jest wypełnienie wartościami tablicy $Q[][]$.

Konstrukcja ciągu LCS

Napiszemy teraz funkcję `LCS_print()`, która w oparciu o zawartość tablicy $Q[][]$ wypisze na ekranie ciąg LCS, a dokładniej *przykładowy* ciąg LCS.

Zaczniemy od prawego dolnego rogu tablicy i będziemy podążać za strzałkami. Ilekroć napotkamy na strzałkę po skosie (`NW`), tylekroć wypiszemy odpowiedni znak z ciągu s[§] i wywołamy rekurencyjnie naszą funkcję od miejsca wskazywanego przez strzałkę. Jeśli będzie inna strzałka, nic nie wypisujemy, tylko posuwamy się zgodnie z kierunkiem strzałki.

Kiedy dojdziemy do brzegu tabeli (lewego lub górnego), wtedy działanie funkcji się kończy.

Brzmi nieźle, ale coś jest nie tak... No właśnie, dostaniemy LCS od tylca, ze znakami wypisanymi w odwrotnej kolejności. Aby otrzymać prawidłową kolejność, należy zamienić miejscami wypisywanie i wywołanie kolejnej instancji funkcji (dlaczego?).

Nie przedłużajmy sprawy, oto kod funkcji:

```
void LCS_print(string &s, vector<vector<int>> &Q, int i, int j)
{
  if(i == 0 || j == 0) return;
  if(Q[i][j] == NW)
  {
    LCS_print(s, Q, i - 1, j - 1);
    cout << s[i - 1];
  }
  else
    if(Q[i][j] == LEFT)
      LCS_print(s, Q, i - 1, j);
    else
      LCS_print(s, Q, i, j - 1);
}
```

Tym razem nie dokładaliśmy żadnego znaku do ciągu s (stąd wypisywanie znaku o indeksie $i-1$), za to dodaliśmy referencję do parametru s w celu zwiększenia efektywności.[¶]

Dla porządku przytaczamy cały kod programu: z obydwoma opisanymi funkcjami i sprawdzającą funkcją `main()`:

```
#include <bits/stdc++.h>
using namespace std;

#define UP 0
#define LEFT 1
#define NW 2

int LCS_length(string s, string t, vector<vector<int>> &Q)
{
  int n = s.size(), m = t.size();
  vector<vector<int>> P;
  s = '#' + s;
  t = '#' + t;
```

[§]Wystarczy, jeśli funkcja `LCS_print()` otrzyma jako argument jeden z ciągów s lub t, gdyż jej zadaniem jest wypisywanie znaków występujących w obu ciągach.

[¶]Dodanie referencji do argumentów s oraz t w funkcji `LCS_length()` spowodowałoby zmianę oryginalnych wartości ciągów w funkcji wywołującej, co mogłoby być niepożądane.

```
  P.resize(n+1);
  Q.resize(n+1);
  for(int i = 0; i <= n; i++)
  {
    P[i].resize(m + 1);
    Q[i].resize(m + 1);
  }
  for(int j = 1; j <= m; j++)
    for(int i = 1; i <= n; i++)
      if(s[i] == t[j])
      {
        P[i][j] = P[i - 1][j - 1] + 1;
        Q[i][j] = NW;
      }
      else
        if(P[i - 1][j] >= P[i][j - 1])
        {
          P[i][j] = P[i - 1][j];
          Q[i][j] = LEFT;
        }
        else
        {
          P[i][j] = P[i][j - 1];
          Q[i][j] = UP;
        }
  return P[n][m];
}

void LCS_print(string &s, vector<vector<int>> &Q, int i, int j)
{
  if(i == 0 || j == 0) return;
  if(Q[i][j] == NW)
  {
    LCS_print(s, Q, i - 1, j - 1);
    cout << s[i - 1];
  }
  else
    if(Q[i][j] == LEFT)
      LCS_print(s, Q, i - 1, j);
    else
      LCS_print(s, Q, i, j - 1);
}

int main()
{
  string s, t;
  cin >> s >> t;
  vector<vector<int>> Q;
  int d = LCS_length(s, t, Q);
  int n = s.size(), m = t.size();
```

```
  LCS_print(s, Q, n, m);
}
```

Po uruchomieniu programu i wpisaniu danych:

```
BBCC
ABCBAC
```

otrzymujemy wynik:

```
BBC
```

Ciąg `BCC` także jest najdłuższym wspólnym podciągiem dla powyższych danych.