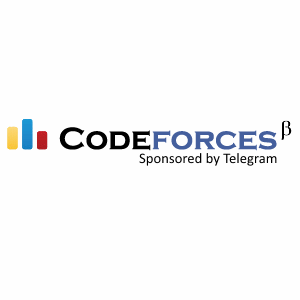
****

**Programação Competitiva**





Diogo Filipe Pinto Pereira

up201605323

**UVA 1125­­8 – String Partition**

**Problema:**  Neste problema, é-nos pedido para, dada uma *string* contendo apenas números inteiros, dividir a *string* de maneira a que a soma dos números obtidos nas divisões seja máxima.

**Complexidade Temporal:** O(n^2), sendo n o tamanho da string.

Para cada posição da *string* a analisar, vamos percorrer a string a partir dessa posição até ao fim, o que significa que a complexidade temporal vai ser O(n^2).

**Complexidade Espacial:** O(n). Neste programa apenas é usado um vetor de *long long int* com n+5 posições que tem complexidade espacial O(N) e uma *string* que é O(N), sendo por isso a complexidade espacial O(n).

1. #include <iostream>
2. #include <cstdlib>
3. #include <vector>
4. #include <string>
5. #include <climits>
7. **using** **namespace** std;
9. **void** calculate(string s) {
10. **int** n = (**int**) s.length();
11. **long** **long** **int** sum = 0;
13. vector<**long** **long** **int**> dp(n+5);
15. **for**(**int** i = 0; i < n; i++) {
16. sum = 0;
17. **for**(**int** j = i; j < n; j++) {
18. sum = sum\*10 + (**int**) s[j]-'0';
19. **if**(sum > INT\_MAX) **break**;
20. dp[j+1] = max(dp[j+1],dp[i] + sum);
21. }
22. }
24. printf("%lld\n", dp[n]);
25. }
27. **int** main() {
28. **int** n;
29. cin >> n;
31. **for**(**int** i = 0; i < n; i++) {
32. string s;
33. cin >> s;
35. calculate(s);
36. }
38. **return** 0;
39. }

**Explicação:**  De maneira a encontrar a solução de forma mais eficiente foi usada programação dinâmica. A ideia é que para cada posição da *string* temos guardado a soma máxima possível.

Comecei por percorrer a *string* completa e para cada posição atual i, percorro a string a partir daí e para cada posição intermédia j vejo qual é que é o número que obtemos. Depois somo esse número ao número que obtivemos em dp[i], que nos indica a soma máxima até então. Se a soma dos dois for maior do que o valor que tenho até agora para essa posição j atualizamos o seu valor no vetor dp. A única precaução a ter é garantir que o número construído não seja maior do que o número máximo representável num inteiro.

**CodeForces 346B – Lucky Common Subsequence**

**Problema:**  O objetivo deste problema é: dado três *strings* encontrar e imprimir a maior subsequência comum entre as duas primeiras, tal que não contenha a terceira (*virus*) como *substring*.

**Complexidade Temporal:** O(n\*m\*k), sendo m e n o tamanho das primeiras duas *strings* e k o tamanho da *string* *virus*.

As funções failure e KMP são ambas O(k), e a função lucky vai ser O(n\*m\*k), pois para cada posição das primeiras duas *strings* precisamos de percorrer também a *string* *virus* para ver se a *string* obtida até então é *substring* da *string* *virus*.

**Complexidade Espacial:** O(n^3), sendo n o tamanho máximo das strings. São usados dois vetores tridimensionais, sendo que cada um ocupa O(n^3), um vetor que ocupa O(n) e três *strings*, ocupando também todas estas O(n), a complexidade espacial vai ser O(n^3).

1. #include <iostream>
2. #include <algorithm>
3. #include <vector>
4. #include <string>
5. #include <cstring>
7. **using** **namespace** std;
9. #define MAX 105
11. vector<**int**> f(MAX);
12. **int** lcs[MAX][MAX][MAX];
13. string lcsstring[MAX][MAX][MAX];
14. string s1, s2,virus;
15. **int** n1, n2, nvirus;
17. **void** failure() {
18. **int** j = -1;
19. f[0] = -1;
21. **for**(**int** i = 1; i < nvirus; i++) {
22. **while**(j > -1 && virus[j+1] != virus[i]) j = f[j];
24. **if**(virus[j+1] == virus[i]) j++;
26. f[i] = j;
27. }
28. }
30. **int** KMP(**int** k, **int** i) {
31. **while**(k >= 0 && virus[k+1] != s1[i]) k = f[k];
33. **if**(virus[k+1] == s1[i])  k++;
35. **return** k;
36. }
38. **int** lucky(**int** i, **int** j, **int** k) {
39. **if**(i == n1 || j == n2) **return** lcs[i][j][k] = 0;
41. **if**(lcs[i][j][k] != -1) **return** lcs[i][j][k];
43. lcs[i][j][k] = 0;
45. **if**(lucky(i,j+1,k) > lucky(i+1,j,k)) {
46. lcs[i][j][k] = lcs[i][j+1][k];
47. lcsstring[i][j][k] = lcsstring[i][j+1][k];
48. }
49. **else** {
50. lcs[i][j][k] = lcs[i+1][j][k];
51. lcsstring[i][j][k] = lcsstring[i+1][j][k];
52. }
54. **if**(s1[i] == s2[j]) {
55. **int** m = KMP(k-1,i);
56. **if**(m != nvirus - 1) {
57. **if**(lucky(i+1,j+1,m+1) + 1 > lcs[i][j][k]) {
58. lcs[i][j][k] =  lcs[i+1][j+1][m+1] + 1;
59. lcsstring[i][j][k] = lcsstring[i+1][j+1][m+1] + s1[i];
60. }
61. }
62. }
64. **return** lcs[i][j][k];
65. }
67. **int** main() {
68. cin >> s1 >> s2 >> virus;
69. n1 = s1.length();
70. n2 = s2.length();
71. nvirus = virus.length();
73. memset(lcs,-1,**sizeof** lcs);
75. failure();
77. **int** ans = lucky(0,0,0);
79. **if**(ans == 0) cout << ans << endl;
80. **else**{
81. reverse(lcsstring[0][0][0].rbegin(),lcsstring[0][0][0].rend());
82. cout << lcsstring[0][0][0] << endl;
83. }
85. **return** 0;
86. }

**Explicação:**  Inicialmente, tentei usar apenas um vetor bidimensional, e usando o algoritmo para encontrar a *longest commom substring*, cada vez que uma letra fosse igual nas duas *strings*, apenas verificava se ao adicionar essa nova letra à *longest commom substring* até então se a *string* originada iria resultar na *string virus*. No entanto este método falhava nos casos em que a *string* *virus* era prefixo das duas *strings*.

Mudei depois o programa para a solução final. Continuei a usar o algoritmo de *longest commom substring*, no entanto desta vez usei um vetor tridimensional. Isto permite-me manter informado sobre a posição atual que nos encontramos na *string* *virus*. Assim, cada vez que encontramos uma letra igual nas duas *strings*, primeiro chamamos o KMP, para verificar que se ao adicionarmos a letra atual, a *string* que vamos obter não vai conter a *string* *virus* como *substring*. No fim, como ao longo do algoritmo, em cada passo, eu ia construindo a *string*, basta reverter a *string*, visto que a *string* obtida estava por ordem inversa, dado que foi obtida de maneira recursiva.

**UVA 976 – Bridge Building**

**Problema:**  Neste problema, é-nos dado uma matriz que representa o mapa, com tamanho RxC, sendo R o número de linhas e C o número de colunas. A matriz é constituída por “**#**”e por “**.”**. O primeiro símbolo representa terra e o segundo representa água. O objetivo é construir B pontes, onde a soma do comprimento de todas elas seja mínima. A única restrição é que entre as diferentes pontes deve existir uma distância mínima de S colunas.

**Complexidade Temporal:** O(R\*C), sendo R o número de linhas e C o número de colunas.

A função *floodfill*, tem complexidade O(R\*C), visto que vamos ter que passar por todas as posições da matriz.

A função calculate tem complexidade O(C\*B), visto que vamos ter que percorrer todas as soluções possíveis, estando alguns valores intermédios já guardados no vetor dp.

Sendo assim, a complexidade total vai ser O(R\*C+C\*B), no entanto como R é maior do que B, a complexidade arredonda para O(R\*C).

**Complexidade Espacial:** O(n^2), sendo n o número máximo de colunas e linhas. São usados quatro vetores de vetores, dois que ocupam O(n^2), outro que ocupa O(n\*m), sendo m o número máximo de pontes., e por último um que ocupa O(2). Como m\*n, é menor que n^2, pois m é no máximo 105, a complexidade espacial vai ser O(n^2).

1. #include <cstdio>
2. #include <vector>
3. #include <climits>
4. #include <algorithm>
5. #include <cstdlib>
6. #include <string>

9. #define MAX 10\*10\*10\*10\*10\*10\*10\*10
10. #define MAXR 10005
11. #define MAXB 105
12. **using** **namespace** std;
14. vector<vector<**int**>> limits(2);
15. **bool** visited[MAXR][MAXR];
16. vector<vector<**int**>> map(MAXR, vector<**int**>(MAXR));
17. vector<vector<**int**>> dp(MAXR, vector<**int**>(MAXB));
19. **int** r, c, b, s;
21. **void** floodfill(**int** x, **int** y, **int** pos) {
22. visited[x][y] = **true**;
24. **if**(pos == 0)
25. limits[pos][y] = max(limits[pos][y],x);
26. **else**
27. limits[pos][y] = min(limits[pos][y],x);
29. **if**(x-1>=0)
30. **if**(map[x-1][y] == 1 && !visited[x-1][y]) floodfill(x-1,y,pos);
32. **if**(y-1 >= 0)
33. **if**(map[x][y-1] == 1 && !visited[x][y-1]) floodfill(x,y-1,pos);
35. **if**(y+1 < c)
36. **if**(map[x][y+1] == 1 && !visited[x][y+1]) floodfill(x,y+1,pos);
38. **if**(x+1 < r)
39. **if**(map[x+1][y] == 1 && !visited[x+1][y]) floodfill(x+1,y,pos);
40. }
42. **int** calculate(**int** pos, **int** b2) {
43. **if**(pos < 0) {
44. **if**(b2 == 0) **return** b2;
45. **else** **return** MAX-5;
46. }
48. **if**(dp[pos][b2] != MAX) **return** dp[pos][b2];
50. **if**(b2 == 0)
51. dp[pos][b2] = 0;
52. **else**
53. dp[pos][b2] = min(dp[pos][b2], min(calculate(pos-s-1,b2-1)+limits[1][pos]-limits[0][pos]-1,calculate(pos-1,b2)));
55. **return** dp[pos][b2];
56. }
58. **int** main() {
59. **while**(scanf("%d %d %d %d", &r,&c,&b,&s) != EOF) {
61. **for**(**int** i = 0; i < r; i++) {
62. fill(map[i].begin(),map[i].end(),0);
63. **char** op[c+5];
64. scanf("%s", op);
65. **for**(**int** j = 0; j < c; j++) {
66. **if**(op[j] == '#') map[i][j] = 1;
67. visited[i][j] = **false**;
68. }
69. }
71. limits[0].resize(c+5);
72. limits[1].resize(c+5);
74. **for**(**int** i = 0; i < c; i++) {
75. fill(dp[i].begin(),dp[i].end(),MAX);
76. limits[0][i] = 0;
77. limits[1][i] = r;
78. }
80. floodfill(0,0,0);
81. floodfill(r-1,0,1);
83. **int** ans = calculate(c-1,b);
85. printf("%d\n", ans);
87. }
89. **return** 0;
90. }

**Explicação:** Para encontrar os limites das margens, fiz dois *flood-fills*, que vão guardar no vetor “*limits*” a margem superior (posição 0) e inferior (posição 1) em cada coluna. Depois de termos os limites das margens chamamos a função “*calculate*”, que vai percorrer as colunas e o número de pontes. Para cada posição do vetor, temos duas hipóteses: colocar a ponte e chamar a função recursivamente para a posição S posições depois, ou então não colocar a ponte nessa coluna e chamar a função de maneira recursiva na posição a seguir. Calculamos o valor dessas duas hipóteses e guardamos na posição o valor mínimo entre as duas.