Programação Dinâmica 2D
Professor Wladimir

Maximum Path Sum

By starting at the top of the triangle below and moving to adjacent numbers on the row below, the maximum total from top to bottom is 23.

$$\begin{array}{ccccccc} & & & \mathbf{3} & & & \\ & & \mathbf{7} & & 4 & & \\ & 2 & & \mathbf{4} & & 6 & \\ 8 & & 5 & & \mathbf{9} & & 3 \end{array}$$

That is, $3+7+4+9 = 23$.

Find the maximum total from top to bottom of the triangle below:

```
75
95 64
17 47 82
18 35 87 10
20 04 82 47 65
19 01 23 75 03 34
88 02 77 73 07 63 67
99 65 04 28 06 16 70 92
41 41 26 56 83 40 80 70 33
41 48 72 33 47 32 37 16 94 29
53 71 44 65 25 43 91 52 97 51 14
70 11 33 28 77 73 17 78 39 68 17 57
91 71 52 38 17 14 91 43 58 50 27 29 48
63 66 04 68 89 53 67 30 73 16 69 87 40 31
04 62 98 27 23 09 70 98 73 93 38 53 60 04 23
```

NOTE: As there are only routes, it is possible to solve this problem by trying every route. However, Problem 67, is the same challenge with a triangle containing one-hundred rows; it cannot be solved by brute force, and requires a clever method! ;o)

Referência:

- Problem 18: Maximum Path Sum I https://projecteuler.net/problem=18
- Problem 67: Maximum Path Sum II https://projecteuler.net/problem=67

Solução

A Figura 1 mostra que um triângulo de altura 4 possui 8 caminhos possíveis, partindo do vértice superior até a base. De forma geral, um triângulo de altura N possui 2^{N-1} caminhos distintos.

Para determinar o caminho de soma máxima, podemos adotar uma abordagem dinâmica. A ideia é que a soma máxima até a linha h pode ser obtida a partir das somas máximas já calculadas até a linha $h-1$. Dessa forma, ao chegar na linha h (começando do zero), é necessário calcular apenas $h+1$ valores de somas máximas, correspondentes a cada posição daquela linha.

A Figura 2 exemplifica esse processo para o triângulo apresentado na Figura 1.

Seja `valor[i][j]` o valor armazenado no nó da i-ésima linha e j-ésima coluna, e seja `table[i][j]` a soma máxima acumulada até esse ponto. Podemos definir a recorrência da seguinte forma:

$$\texttt{table[0][0]} = \texttt{valor[0][0]}$$
$$\texttt{table[i][0]} = \texttt{table[i-1][0]} + \texttt{valor[i][0]}$$
$$\texttt{table[i][i]} = \texttt{table[i-1][i-1]} + \texttt{valor[i][i]}$$
$$\texttt{table[i][j]} = \max(\texttt{table[i-1][j-1]}, \texttt{table[i-1][j]}) + \texttt{valor[i][j]} \quad \text{para } 0 < j < i$$

O caminho de soma máxima será dado por

$$\texttt{SOMA\_MAX} = \min_{k \in \{0,\ldots,N-1\}} \{\texttt{table}[N-1][k]\}$$

O código que calcula a recorrência é o seguinte:

```cpp
#include <iostream>
#include <algorithm>    // std::max

using namespace std;
int main()
{
   int i, j, N;
   scanf("%d", &N);
   int valor[N][N];
   int table[N][N];

   for(int i = 0; i < N; i++){
       for(int j = 0 ; j <= i; j++){
           scanf("%d", &valor[i][j]);
       }
   }

   table[0][0] = valor[0][0];
   for(int i = 1; i < N; i++){
       for(j = 0; j <= i; j++){
        if(j==0) table[i][j] = table[i-1][j] + valor[i][j];
        else if(j == i) table[i][j] = table[i-1][j-1] + valor[i][j];
        else table[i][j] = max(table[i-1][j-1], table[i-1][j]) + valor[i][j];
       }
   }

   int resp = 0;
   for(int j = 0; j < N; j++){
       resp = max(resp, table[N-1][j]);
   }

   printf("resp %d\n", resp);
}
```

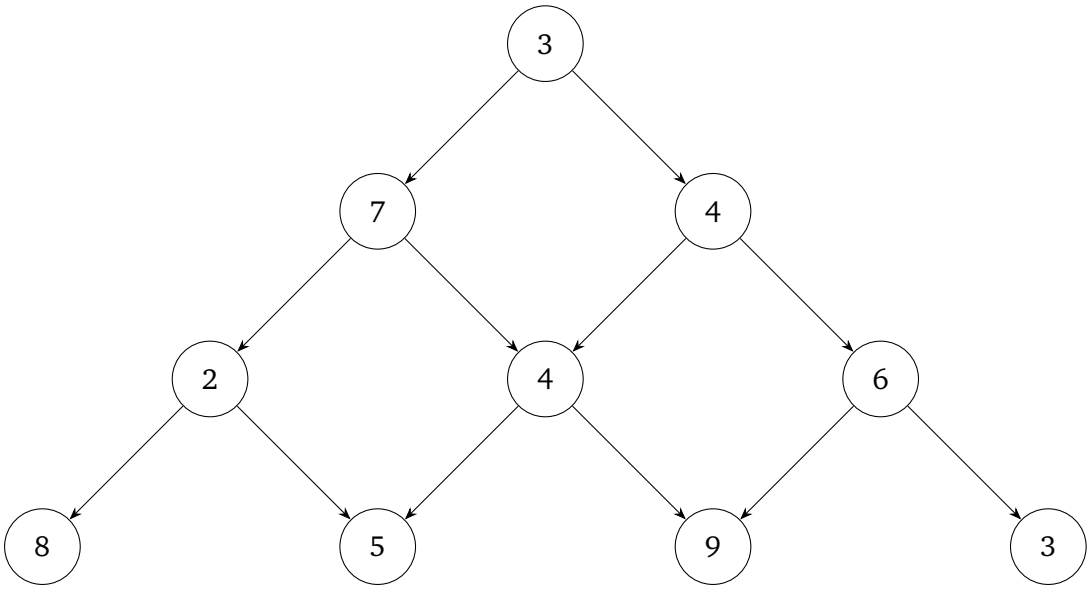

Figura 1: Rotas no triângulo de altura 4

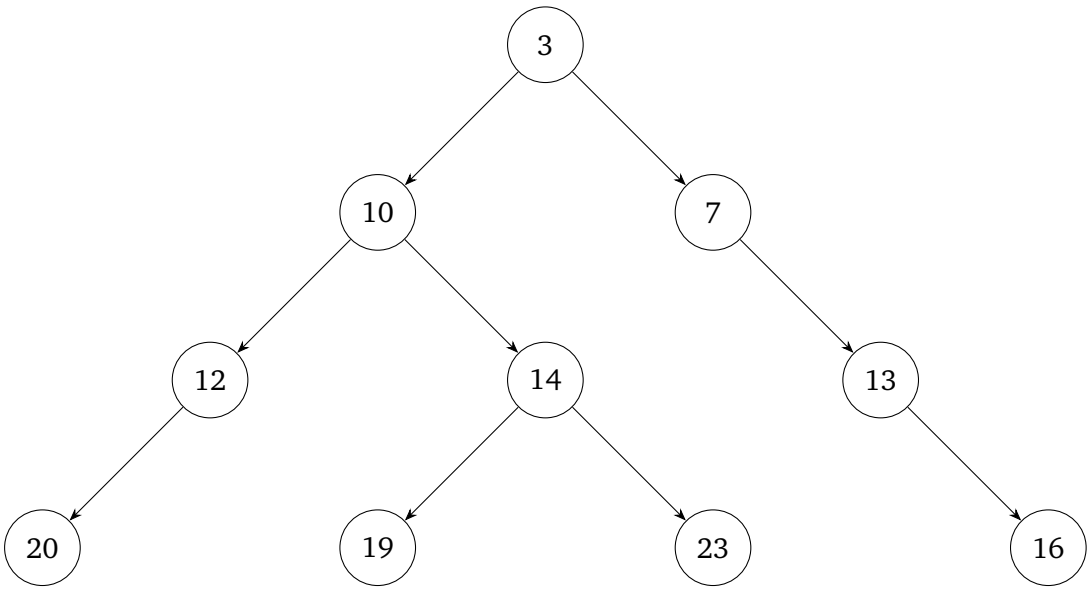

Figura 2: Cálculo dos caminhos de soma máxima

Problema: Minimização do Produto Energia × Tempo (EDP)

Dado um conjunto de programas que devem ser executados sequencialmente em um laptop, deseja-se determinar o nível de frequência ideal de execução para cada programa, de forma a minimizar o *Energy* × *Delay Product* (EDP) total.

O laptop possui F níveis de frequência numerados de 1 a F. Cada programa pode ser executado em qualquer um desses níveis, e para cada par (p, f) (programa p no nível de frequência f), são fornecidos:

- $E_{p,f}$: energia consumida em Joules;
- $A_{p,f}$: tempo de execução em milissegundos.

Além disso, há um custo fixo de troca entre frequências:

- E: energia necessária (Joules) para mudar entre quaisquer dois níveis;
- A: tempo necessário (ms) para tal troca.

O processador começa no nível de frequência 1.

$F = 2$

$P = 3$

$E = 10$

$A = 10$

	P=1	**P=2**	**P=3**
F=1	6.000	300.000	400.000
F=2	9.000	300.000	350.000

Figura 3: Exemplo com 2 níveis de frequência e 3 programas

Referência:

- Maratona SBC 2006 - Energia × Tempo https://judge.beecrowd.com/pt/problems/view/3438

Solução

Podemos observar novamente que o custo mínimo para executar até o i-ésimo programa na j-ésima frequência pode ser calculado a partir dos custos mínimos para executar até o $(i-1)$-ésimo programa em todas as frequências disponíveis. Dessa forma, para cada programa, é necessário calcular o custo mínimo associado a cada frequência.

Seja `custo[i][j]` o custo em EDP para executar até o i-ésimo programa na frequência j, e seja `table[i][j]` o custo mínimo acumulado em EDP até o i-ésimo programa na frequência j. A relação de recorrência que define a solução é dada por:

$$\begin{aligned}
\texttt{table}[0][0] &= \texttt{custo}[0][0] \\
\texttt{table}[0][j] &= \texttt{custo}[0][j] + E \cdot A, \quad \text{para } j > 0 \\
\texttt{table}[i][j] &= \min_{k \in F} \{\texttt{table}[i-1][k] + E \cdot A \cdot [k \neq j]\} + \texttt{custo}[i][j]
\end{aligned}$$

onde:

- E representa o custo de transição entre frequências;
- A é um fator de penalização associado à troca de frequência;
- F é o conjunto de todas as frequências disponíveis;
- $[k \neq j]$ é 1 se $k \neq j$, e 0 caso contrário (função indicadora).

O EDP mínimo será dado por

$$\texttt{EDP\_MIN} = \min_{k \in F} \{\texttt{table}[P-1][k]\}$$

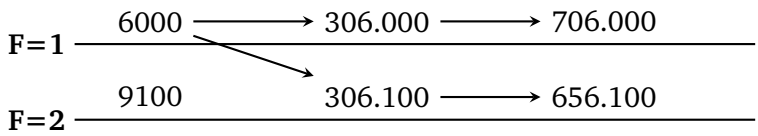

Figura 4: Cálculo do EDP mínimo

Maior Subsequência Comum

Dadas duas palavras s_1 e s_2, dizemos que s_1 é uma **subsequência** de s_2 se é possível obter s_1 a partir de s_2 removendo alguns (possivelmente nenhum) caracteres, sem alterar a ordem dos caracteres restantes.

Por exemplo, `AAT` é uma subsequência de `AGAT`, pois podemos remover o 'G' e obter `AAT`. No entanto, `AAT` não é uma subsequência de `AGTA`, pois a ordem dos caracteres precisa ser preservada.

Dado isso, nosso objetivo é, dadas duas palavras s_1 e s_2, encontrar a **maior subsequência comum** entre elas, isto é, determinar o comprimento da maior sequência que é subsequência de ambas as palavras simultaneamente.

Referência:

- Olimpíada Brasileira de Informática – OBI2013 - Parque Jurássico https://judge.beecrowd.com/pt/problems/view/3045

Solução

Denotaremos por $s_1[0..j]$ a subpalavra de s_1 que vai do caractere de índice 0 até o caractere de índice j, inclusive. A função $LCS(s_1[0..j], s_2[0..k])$, que representa o tamanho da maior subsequência comum entre os prefixos $s_1[0..j]$ e $s_2[0..k]$, pode ser definida recursivamente da seguinte forma:

- Se $s_1[j] = s_2[k]$, então esse caractere pertence à maior subsequência comum, e o problema se reduz a calcular $LCS(s_1[0..j-1], s_2[0..k-1])$.
- Se $s_1[j] \neq s_2[k]$, então o caractere atual não contribui para a subsequência comum, e precisamos considerar os dois casos recursivos: $LCS(s_1[0..j-1], s_2[0..k])$ e $LCS(s_1[0..j], s_2[0..k-1])$, escolhendo o maior entre eles.

	ε	B	D	C	A	B	C
ε	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	2
C	0	1	1	2	2	2	3
B	0	1	1	2	2	3	3
D	0	1	2	2	2	3	3
A	0	1	2	2	3	3	3
B	0	1	2	2	3	4	4

Figura 5: Calculando a maior subsequência comum de s1 = "BDCABC"e s2 = "ABCBDAB"@

Implementação

```
int D[1001][1001];

n = strlen(x);
m = strlen(y);

for(int i = 0; i <= n; i++) D[i][0] = 0;
for(int j = 0; j <= m; j++) D[0][j] = 0;

for(int i = 1; i <= n; i++)
```

```
    for(int j = 1; j <= m; j++)
        if( s1[i-1] == s2[j-1] )
            D[i][j] = D[i-1][j-1] + 1;
        else
            D[i][j] = max(D[i-1][j], D[i][j-1]);
```

Reduzindo o uso de memória

```
int D[2][1001];
int ans;
n = strlen(x);
m = strlen(y);

for(int j = 0; j <= m; j++) D[0][j] = 0;

for(int i = 1; i <= n; i++){
    int ii = i&1;
    D[ii][0] = 0;
    for(int j = 1; j <= m; j++){
        if( x[i-1] == y[j-1] )
            D[ii][j] = D[1-ii][j-1] + 1;
        else
            D[ii][j] = max(D[1-ii][j], D[ii][j-1]);
    }
}
ans = D[n&1][m];
```