Programação Dinâmica

Amanda de Souza
Daniel Marques
Rafael Assis

Definição de Programação Dinâmica

- **Programação Dinâmica:** Utiliza o método de “dividir e conquistar” para resolver um problema através da combinação de subproblemas. Cada subproblema é resolvido somente uma vez, e sua resposta é armazenada para ser usada nos subproblemas subsequentes.

- As ideias centrais são:
  - 1. Resolver subproblemas mais simples (de menor grau)
  - 2. Memorização: armazenar as soluções dos subproblemas resolvidos.
  - 3. Combinar: juntar soluções dos subproblemas para formar a solução de subproblemas cada vez maiores (de maior grau) até se obter a solução do problema requisitado.

- Vantagens da programação dinâmica:
  - Opta tempo por espaço (*trade-off).*
    - Por armazenar valores (espaço)
    - Por evitar resolver um mesmo subproblema várias vezes (tempo).

Nome “programação dinâmica”

- Ian Parberry (Problems on Algorithms, Prentice Hall, 1995.)

  “*A programação dinâmica é um nome sofisticado para recursão com uma tabela. Em vez de resolver subproblemas recursivamente, resolva-os sequencialmente e armazene suas soluções em uma tabela. O truque é resolvê-los na ordem correta para que sempre que a solução para um subproblema seja necessária, ela já esteja disponível na tabela. A programação dinâmica é particularmente útil em problemas para os quais dividir e conquistar parece gerar um número exponencial de subproblemas, mas na verdade há apenas um pequeno número de subproblemas repetidos exponencialmente com frequência. Nesse caso, faz sentido computar cada solução na primeira vez e armazená-las em uma tabela para uso posterior, em vez de recalcula-lás recursivamente toda vez que forem necessárias.*”

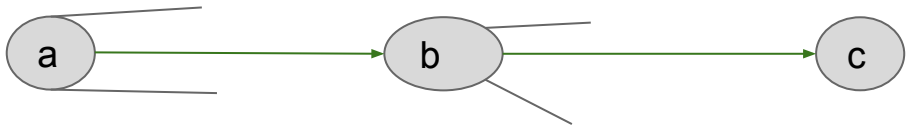
- Para Richard Bellman, seu criador, é apenas um nome chique, então pode ser resumido como recursão com tabela.

PD em Otimização

- PD é aplicada a problemas de otimização: Um problema que possui várias soluções com qualidades diferentes e deseja-se encontrar a solução ótima.
    - Costumam ser problemas NP-Hard consumindo muito recursos por terem complexidade não polinomial.
    - Então, usa-se PD para melhorar o processo.

- Princípio de Otimização de Bellman:
  "Uma trajetória ótima tem a seguinte propriedade: quaisquer que tenham sido os passos anteriores, a trajetória remanescente deverá ser uma trajetória ótima com respeito ao estado resultante dos passos anteriores, ou seja, uma política ótima é formada de sub-políticas ótimas."

Exemplo: Seja P(a,c) um caminho entre os nós *(a,c)*, P(a,c) será uma solução ótima se for composto pelas soluções ótimas dos seus subproblemas, no caso, se tiver as soluções ótimas para os nós (b,c) e (a,b).



Como e quando usar PD para otimização

- Quando o problema tem uma subestrutura ótima:
  - Quando se identifica e caracteriza a estrutura de uma solução ótima como sendo compostas por soluções ótimas de subproblemas, ou seja, quando seu problema respeita o princípio de Bellman.

- Quando os Subproblemas são superpostos:
  - Quando se identifica que o espaço de subproblemas é pequeno, no sentido que, um algoritmo recursivo resolve várias vezes os mesmos subproblemas em subproblemas e momentos distintos.
  - Em geral, o número de subproblemas é da mesma ordem da entrada:
    - Ou seja, dado uma entrada n, há cerca de n subproblemas distintos a serem resolvidos
    - O problema é que: um mesmo problema é resolvido várias vezes em situações diferentes o que pode ser melhorado com PD.

Estratégia de PD

- Ao desenvolver um algoritmo de PD, seguimos uma sequência de passos:

    1. Caracterizar a estrutura de uma solução ótima.
    2. Definir recursivamente o valor de uma solução ótima em funções de soluções ótimas de subproblemas.
    3. Calcular as soluções ótimas dos subproblemas, normalmente de forma ascendente (bottom-up) ou descendente (top-down).
    4. Construir uma solução ótima a partir das soluçẽos dos subproblemas encontradas anteriormente.

Estratégia

A principal estratégia da programação dinâmica é **armazenar** o resultado dos subproblemas resolvidos.

Exemplo Fatorial:

Fat = [**F(0)**] → F(n) calcula fatorial de n

Fat[1] = 1 . Fat[0]

Fat[2] = 2 . Fat[2]

Fat[3] = 3 . Fat[2]

Fat[n] = n . Fat[n-1] → Fat[n] armazena fatorial de n

Abordagens

A programação dinâmica usa de duas abordagens para resolver e otimizar problemas, sendo elas:

- **Top-Down**
  - Encontra a solução para a maior instância **n** usando todos os subproblemas **n-i** até condição de parada **n - i = p**.
  - Utiliza a estratégia de memoização (um neologismo para a palavra inglês “memo” : gravado para ser usada mais tarde), o que costuma deixar o algoritmo em duas camadas e também recursivo.

- **Bottom-Up**

  - Encontra a solução para a menor instância **p** do problema, calcula todos os subproblemas **p + i** até condição de parada **n = p + 1**.
  - Costumam ser algoritmos semelhante aos Iterativos

Sequência de Fibonacci

Sequência de Fibonacci: uma sequência de números inteiros, começando normalmente por 0 e 1, na qual, cada termo subsequente corresponde à soma dos dois anteriores.

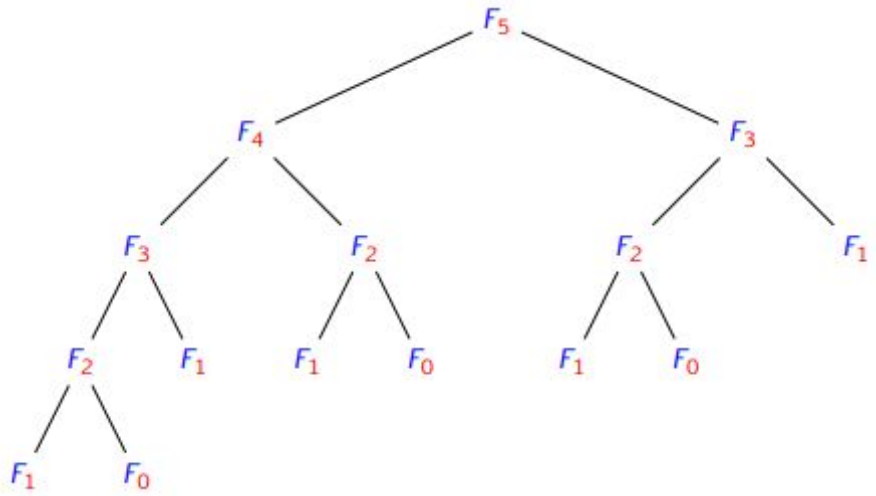
$$F(n) = \begin{cases} 0, & \text{se } n = 0; \\ 1, & \text{se } n = 1; \\ F(n-1) + F(n-2) & \text{outros casos.} \end{cases}$$

Sequência para os 19 primeiros termos:

- 0,1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, ...

Sequência de Fibonacci

- Exemplo: Pela fórmula, sabemos que F(5) = F(4) + F(3); mas não sabemos Nem F(4) e nem F(3). Então tem que calcular (F(4) = F(3) + F(2)) (F(3) = F(2) + F(1)) e também F(2) = F(1) + F(0).!
- Desses, somente F(0) e F(1) temos o valor, a partir de F(2) É necessário calcular os outro F(N)
- Observe nas imagens o seguinte caso: Para calcular F(4) e F(3) foi necessário calcular F(2) e isso foi feito mais de uma vez (indício que pode usar PD: Subproblemas Sobrepostos).




```
FIBO-REC(5) = 5
FIBO-REC(5)
  FIBO-REC(4)
    FIBO-REC(3)
      FIBO-REC(2)
        FIBO-REC(1)
        FIBO-REC(0)
      FIBO-REC(1)
    FIBO-REC(2)
      FIBO-REC(1)
      FIBO-REC(0)
  FIBO-REC(3)
    FIBO-REC(2)
      FIBO-REC(1)
      FIBO-REC(0)
    FIBO-REC(1)
```

Sequência de Fibonacci

- Perceba que, para resolver um F(n) qualquer, varios i = 1,2 .... , n serão calculados várias vezes.
- Esse problema pode ser melhorada a partir de PD
  - Guarda-se os valores de F(i) quando calculados.
  - Quando necessários, pega-os da tabela.

Sequência de Fibonacci

```
// Estratégia Bottom-Up
int fibonacci(int n) {
    fib[0] = 0;
    fib[1] = 1;
    int i;
    for (i = 2; i < n; i++){
        fib[i] = fib[i-1] + fib[i-2];
     }
     return fib[n];
}
```

Sequência de Fibonacci

```
// Estratégia Top-Down
int memo_fibonacci(n){
    int fib[n];
    for(int i = 0; i < n; i++){
        fib[i] = -1;
    }
    return fib_recursivo(fib, n);
}
```

```
int fib_recursivo(int* fib, int n){
    // Valor F(n) já calculado
    if(fib[n] >= 0){
        return fib[n];
    }
    // caso iniciais: de F(0) e F(1)
    if(n <= 1){
        fib[n] = n;
    }else{
     // A partir de F(2) calcular novos F(n)
     fib[n] = fib_recursivo(fib, n -1) +
          fib_recursivo(fib, n - 2);
    }
    return f[n];
}
```

Problema do Corte da Haste

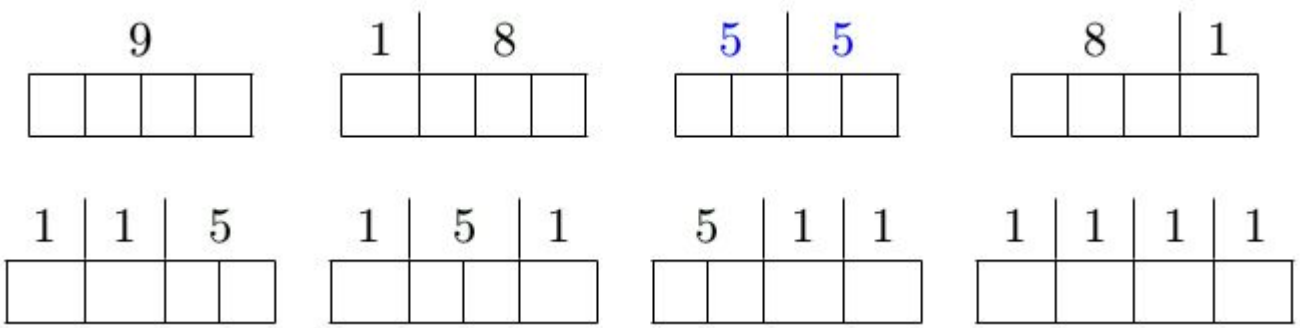
Descrição do Problema do Corte de Haste:

- Dada uma haste de comprimento ***n*** polegadas e uma tabela de preços **p** onde p_i para i = 1, 2, ..., n, determinar a receita máxima $\mathbf{r_n}$ que se pode obter ao se cortar a haste e vender os seus pedaços. Os comprimentos dos pedaços são sempre números inteiros.
- Segue a tabela **p** de preços das hastes de tamanho **n** inteiras

comprimento n	1	2	3	4	5	6	7	8	9	10
preço p_n (R$)	1	5	8	9	10	17	17	20	24	30

Problema do Corte da Haste

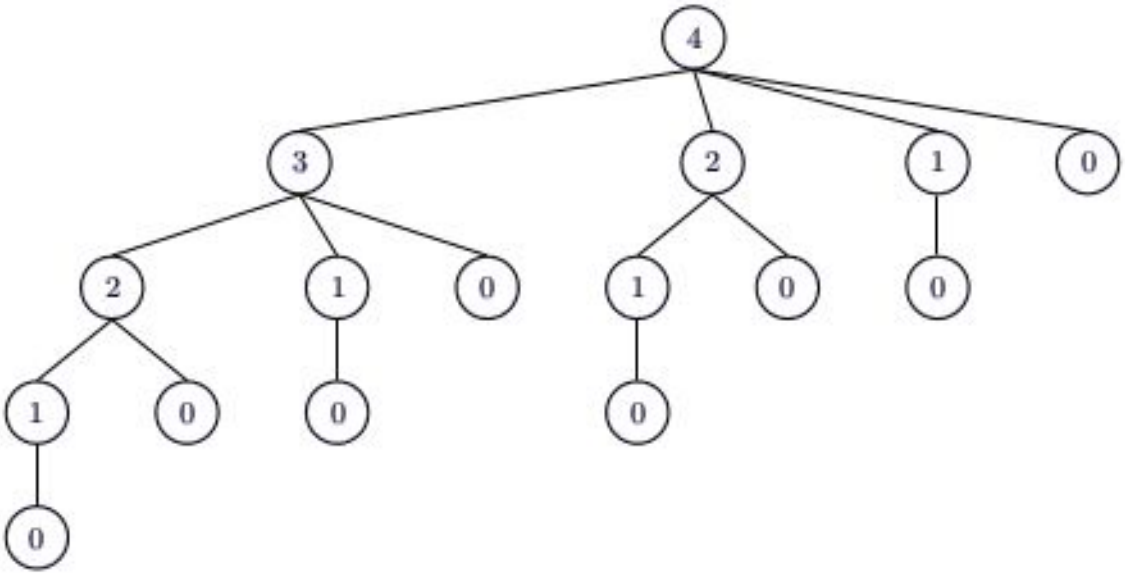
- Como cortar uma Haste de 4 polegadas de forma a se obter a maior receita possível?
  - Para 4 polegadas, há 8 formas



- A maior receita é obtida a do corte ao meio, obtendo-se dois pedaços ($p_2 + p_2 = 10$).
  - Assim $r_4 = 10$

Problema do Corte da Haste

- Uma ideia inicial para resolver esse problema seria:
  - “Calcular todas as possibilidades e compará-las”.
  - Então, poderíamos formar a seguinte árvore de cálculos para resolver isso:
  - Para cada nó, será calculado o r_n pegando o maior dos filhos.



Problema do Corte da Haste

- Perceba que é um problema de Otimização: há várias soluções possíveis válidas (há 8 formas de corte) mas somente uma que proporciona o maior valor.

- É viável força bruta?
  - Em geral, uma haste de comprimento n pode ser cortada em 2^{n-1} formas diferentes, pois podemos escolher cortar ou não em todas as distâncias i, com $1 \leq i \leq n - 1$, contadas a partir da extremidade esquerda.

- É um problema de otimização e NP-Hard: é fácil verificar uma solução mas não encontrar a melhor de todas.
  - A medida que n cresce, o programa demoraria muito mais.

Problema do Corte da Haste

Podemos encontrar a solução a partir da equação abaixo, onde r_n é o valor da solução ótima para uma haste de n polegadas.

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1)$$

Onde:

- p(n) = preço da haste inteira (não cortada).
- r_i = melhor corte (ou não) para um haste de tamanho i.

- Uma vez realizado o primeiro corte podemos considerar os dois pedaços como instâncias independentes do problema do corte de hastes.

Problema do Corte da Haste

- Para resolver o problema original de comprimento n temos que resolver problemas do mesmo tipo, mas de comprimento menores.

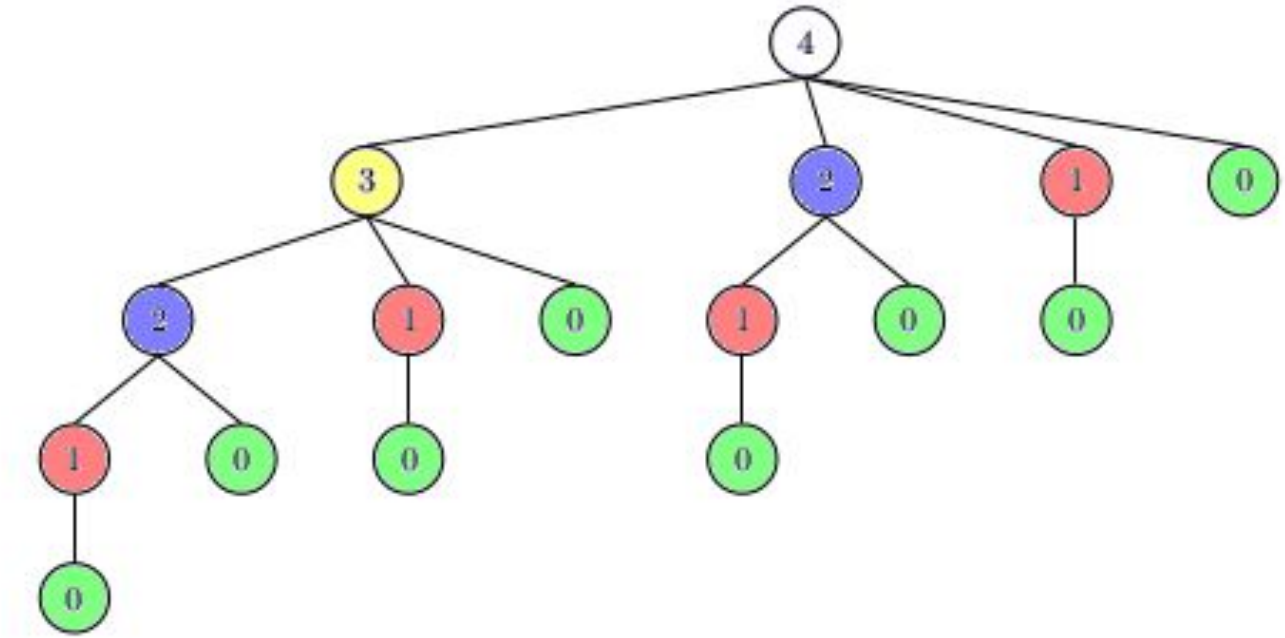
- A solução global ótima incorpora as soluções ótimas de dois subproblemas relacionados, maximizando a receita a partir de cada um desses dois pedaços.

- O problema do corte de hastes exibe subestrutura ótima:
  - **As soluções ótimas para um problema incorporam soluções ótimas para subproblemas relacionados que podem ser resolvidos independentemente.**

- Então, **é possível otimizar esse problema a partir de PD** , assim vai economizar recursos armazenando os resultados dos subproblemas já calculados

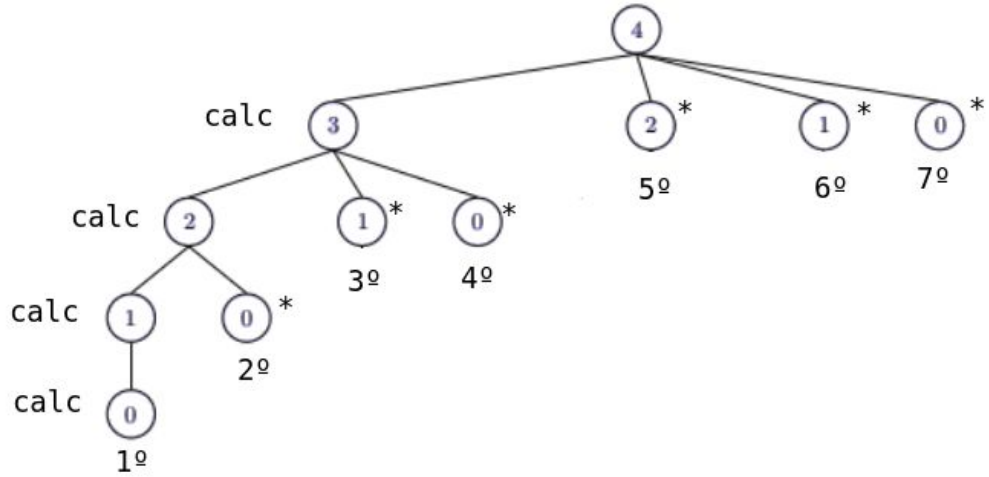
Problema do Corte da Haste

- Marcando as chamadas, perceba que várias vezes é feito um mesmo processamento.
- Aplicar PD irá reduzir o tempo por economiza cálculos já feitos anteriormente.



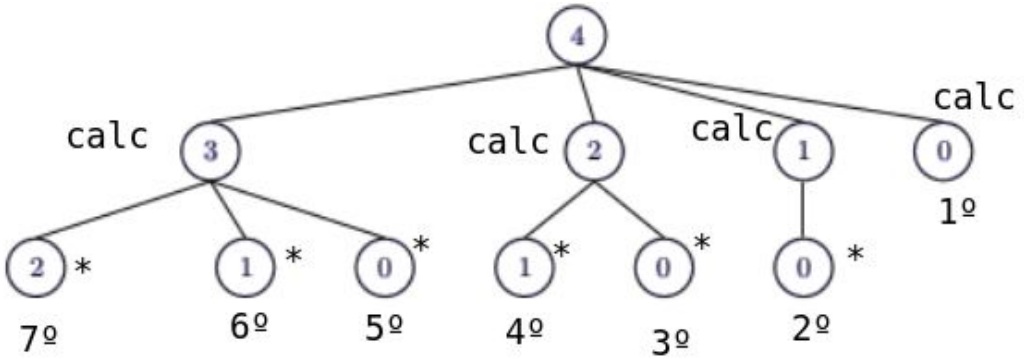
Problema do Corte da Haste

```c
/* Solução Top-Down */
int corte_haste(int *p, int n){
  int r[n+1]; // cria vetor
  r[0] = 0; //caso inicial
  for(int i = 1; i <= n ; i++){
    r[i] = -1; // inicializa valores otimos
  }
  return corte_haste_rec(p, n, r);
}


int corte_haste_rec(int *p, int n, int *r){
 if(r[n] >= 0)// Identifica que o foi calculado r[n] e o usa.
    return r[n];
 int q = -1; // sera calculado um novo r[n]
 for(int i = 1; i <= n; i++){
   q = max(q, p[i] + corte_haste_rec(p, n - i, r));
 }
 r[n] = q; // insere r[n] na tabela para ser usado em outras etapas
 return q;
}
```



Problema do Corte da Haste

```
/* Solução Bottom-Up
p = array de preços onde p[i] é o preço da haste inteira de tamanho i.
n = tamanho da haste que se quer saber seu valor ótimo.
r = array com os valores ótimo, onde r[i] é o valor ótimo para uma haste de tamanho i.
*/

int corte_haste(int *p, int n){
 int r[n+1];
 r[0] = 0;
 int q;
 for(int i = 1; i <= n; i++){
   q = -1;
   for(int j = 1; j <= i; j++){
     q = max(q, p[j] + r[i-j]);
   }
   r[i] = q;
 }
 return r[n];
}
```



Maior Subsequência Comum (LCS)

- Dadas duas sequências X = x1x2 ... xm e Y = y1y2 ... yn, encontrar uma subsequência comum a X e Y que seja o mais longa possível.

- Exemplo:
  - X = n o c t u r n o
  - Y = m o s q u i t e i r o
  - LCS(X,Y) = o t r o (também pode ser o u r o)

- A sequência não deve necessariamente ser contígua!

Maior Subsequência Comum (LCS)

- Algoritmo de força bruta: Gerar todas as subsequências de X e verificar se também é subsequência de Y, e ir guardando a subsequência mais longa vista até o momento.

- Complexidade:
  - O(2^m) para gerar todas as subsequências de X
  - O(n) para verificar se uma subsequência de X é subsequência de Y.
  - Total: O(n2^m)
  - Exponencial!

Maior Subsequência Comum (LCS)

- Aplicando Programação Dinâmica:
  - c[i , j] = c[i - 1, j - 1] + 1, se s1[i] == s2[j]
  - c[i, j] = max(c[i, j - 1], c[i - 1, j]), caso contrário

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

Maior Subsequência Comum (LCS)

```
// Solução Bottom-Up
int get_len_lcs(string & s1, string & s2){
   int len_s1 = s1.size(), len_s2 = s2.size();
   int mat[len_s1+1][len_s2+1];

   for(int i = 1; i <= len_s1; i++){
      mat[i][0] = 0;
   for(int i = 0; i <= len_s2; i++){
     mat[0][i] = 0;
   }
   for(int i = 1; i<=len_s1; i++){
     for(int j = 1; j<=len_s2; j++){
       if(s1[i - 1] == s2[j - 1])
         mat[i][j] = mat[i -  1][j - 1] + 1;
       else
         mat[i][j] = max(mat[i][j -  1], mat[i - 1][j]);
     }
   }
 }
 return mat[len_s1][len_s2];
}
```

Maior Subsequência Comum (LCS)

```
int main(int argc, char *argv[]){
 string s1("ABCB"), s2("BDCAB");
 int len_lcs = get_len_lcs(s1, s2);
  cout << "Length: " << len_lcs << endl;
 return 0;
}
```

Referências

- Livro: Algoritmos: Teoria e Prática. Cormen. 2 ed
- https://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/dynamic-programming.html
- https://pt.wikipedia.org/wiki/Programa%C3%A7%C3%A3o_din%C3%A2mica
- http://www.pucrs.br/ciencias/viali/graduacao/po_2/literatura/pdinamica/monografias/arq0225.pdf
- https://www.ime.usp.br/~am/5711/aulas/aula09.pdf
- http://arcanesentiment.blogspot.com/2010/04/why-dynamic-programming.html
- https://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/NPcompleto.html
- https://pt.wikipedia.org/wiki/Sequ%C3%AAncia_de_Fibonacci
- https://www.ime.usp.br/~cris/aulas/11_1_338/slides/aula13.pdf
- https://gist.github.com/marcoscastro/00456ee20d80c08219ed
- http://edirlei.3dgb.com.br/aulas/paa/PAA_Aula_04_LCS_2015.pdf