Construção e Análise de Algoritmos

aula 22: Dois problemas sobre sequências

1 Introdução

A vida é feita de altos e baixos.

Isso todo mundo sabe.

Mas o otimista gosta mais de ver os altos.

E o pessimista gosta mais de ver os baixos.

Na realidade, a coisa não é bem assim ...

O otimista se sente bem em ver as coisas melhorando.

E o pessimista gosta de se aborrecer vendo as coisas piorando.

Mas, se você pedir a ambos para contar uma história, eles vão usar estratégias muito parecidas.

O otimista vai buscar no passado os momentos mais tristes, e irá montar sua história contando como ele foi pouco a pouco vencendo as dificuldades, para chegar até onde ele está hoje.

O pessimista vai buscar no passado os momentos mais felizes, e irá montar sua história contando como aos poucos a situação foi se deteriorando, até chegar no ponto em que estamos hoje.

Ou seja, tanto um como o outro olham para os altos e os baixos.

Mas eles organizam as coisas de maneira diferente.

Não é engraçado?

Na verdade, isso é uma tremenda ilusão cognitiva ...

O nosso objetivo aqui é ajudar os amigos acima a construir a maior ilusão cognitiva possível.

Nós podemos formalizar o problema da seguinte maneira:

- Nós temos uma sequência de números

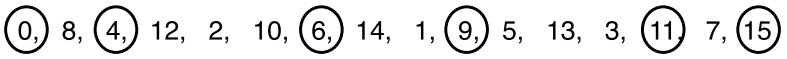
$$a_1,\ a_2,\ a_3,\ \ldots,\ a_{n-1},\ a_n$$

  e a nossa tarefa consiste em selecionar um subconjunto deles (respeitando a ordem)

$$a_{i_1},\ a_{i_2},\ \ldots,\ a_{i_k}$$

  para formar a maior subsequência crescente (ou decrescente) possível.

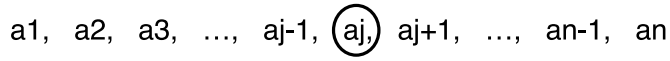
Abaixo nós temos um exemplo simples e uma solução ótima



Pensando em termos de divisão e conquista, nós podemos tentar decompor o problema em dois subproblemas independentes que podem ser resolvidos separadamente.

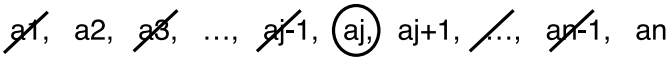
É fácil, veja só.

Suponha que nós selecionamos, por exemplo, o número a_j para fazer parte da solução.



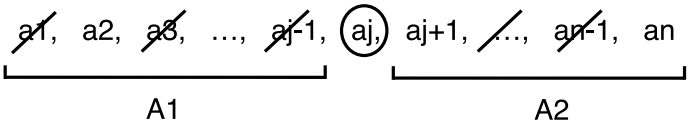
Se isso é o caso, então todo número selecionado no lado esquerdo a partir de agora deve ser menor do que a_j, e todo número selecionado no lado direito deve ser maior que a_j.

Quer dizer, os números que não satisfazem essa condição podem ser eliminados do problema.



Finalmente, basta encontrar subsequências crescentes no lado esquerdo e no lado direito, com o maior tamanho possível.

Mas, esse é o mesmo problema que nós tínhamos no início, só que um pouco menores.



Calculando a solução desses dois subproblemas, e acrescentando a_j entre elas, nós obtemos uma solução para o problema original.

Nesse ponto, alguém pode perguntar: "*Quem disse que a_j faz parte da solução?*"

De fato, nós não temos certeza disso ...

Mas, nós sempre podemos fazer a mesma coisa com todos os outros elementos, e escolher a melhor solução que aparecer.

Abaixo nós temos o pseudo-código do algoritmo (versão não-memoizada, para simplificar as coisas):

```
Procedimento  SC+L-PD1.0  ( A: sequência com n números )
 {
    Se ( A está vazio )   Retorna (vazio)

    Smax  <-- vazio;

    Para  aj  em  A
     {
        (A1,A2)  <--  Quebra (A,aj)

             S1  <--  SC+L-PD1.0 (A1)
             S2  <--  SC+L-PD1.0 (A2)

              S  <--  Concatena (S1,aj,S2)

        Se ( tamanho(S) > tamanho(Smax) )
         {
            Smax  <--  S;
         }
     }
    Retorna (Smax)
 }
```

Análise de complexidade

Como usual na análise de um algoritmo de programação dinâmica, a parte mais importante consiste em determinar o número total de chamadas recursivas.

A observação chave aqui é que cada chamada recursiva corresponde a um subproblema A' que foi construído pela seleção de um elemento a_i à sua esquerda, e um elemento a_j à sua direita:

```
ai  >>   A'= { . . . }   <<  aj
```

Isto é, o subproblema A' consiste na porção da sequência original entre a_i e a_j, depois de eliminados os elementos menores que a_i e maiores que a_j.

(Você consegue ver isso?)

Bom, mas se isso é o caso, então cada subproblema está associado a um par (a_i, a_j).

E, como existem (no máximo) $n \times n$ maneiras de escolher os índices i,j, nós temos um total de $O(n^2)$ subproblemas — e chamadas recursivas.

Legal.

A seguir, nós precisamos estimar o tempo de execução de cada chamada recursiva.

É fácil ver que o laço realiza $O(n)$ iterações, e que a operação de `Quebra` pode ser feita em tempo $O(n)$.

Todo o resto executa em tempo $O(1)$.

Portanto, cada chamada leva tempo $O(n^2)$, e o algoritmo como um tempo executa em tempo

$$O(n^2) \quad \times \quad O(n^2) \quad = \quad O(n^4)$$

Isso não é lá muito bom, mas é melhor que nada (i.e., um algoritmo exponencial).

Estratégia alternativa de decomposição

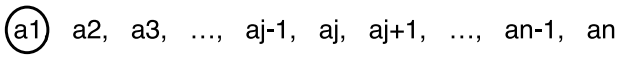
A seguir, nós vamos pensar sobre o problema em termos de algoritmos gulosos.

Quer dizer, nós vamos tentar decompor o problema em um problema só um pouquinho menor, pela seleção de um elemento para a solução.

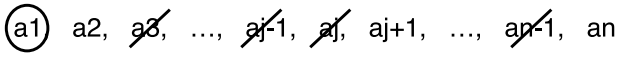
Isso também é fácil, veja só.

O raciocínio guloso seria algo assim: "*Já que você quer a maior subsequência possível, talvez seja uma boa ideia começar logo com o primeiro número.*"

Suponha, portanto, que nós selecinamos o número a_1 para a solução.

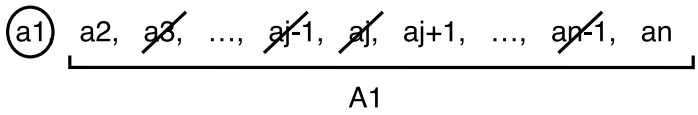


Então, todos os números selecionados a partir de agora devem ser maiores que a_1, e aqueles que não satisfazem essa condição podem ser eliminados do problema.



Finalmente, basta encontrar uma subsequência crescente dentre os números que restaram, com o maior tamanho possível.

Mas, esse é o mesmo problema que nós tínhamos no início, só que um pouquinho menor.

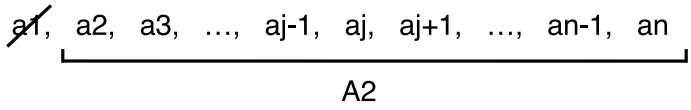


Calculando a solução desse subproblema, e acrescentando a_1 no início, nós obtemos uma solução para o problema original.

Nesse ponto, alguém pode perguntar: "*Quem disse que a_1 faz parte da solução?*"

De fato, nós não temos certeza disso ...

Mas, nós sempre podemos deixar a_1 de lado, fazer tudo de novo com a sequência sem a_1, e depois escolher a melhor solução das duas.



Abaixo nós temos o pseudo-código do algoritmo (versão não-memoizada, para simplificar as coisas):

```
Procedimento  SC+L-PD2.0  ( A: sequência com n números )
 {
    Se ( A está vazio )   Retorna (vazio)

    A1  <--  Remove-e-Elimina (A,a1)     // remove a1 e elimina incompatíveis
    S1  <--  SC+L-PD2.0 (A1)
    S1  <--  Concatena (a1,S1)

    A2  <--  Remove (A,a1)               // só remove a1
    S2  <--  SC+L-PD2.0 (A2)

    Se ( tamanho(S1) > tamanho(S2) )    Retorna (S1)
    Senão                               Retorna (S2)
 }
```

Análise de complexidade

O primeiro passo da análise consiste em determinar o número total de chamadas recursivas.

Mas, dessa vez, todo cuidado é pouco.

À primeira vista, toda chamada recursiva recebe um subproblema da forma $A' = \{a_i, \ldots, a_n\}$.

É fácil ver que existe apenas $O(n)$ subproblemas desse tipo.

Daí, nós poderíamos concluir que são realizadas $O(n)$ chamadas recursivas.

Mas, é preciso lembrar da eliminação de elementos incompatíveis.

Por exemplo, se a sequência original de números é

$$A \;=\; \{\, 10,\, 17,\, 5,\, 11,\, 7,\, 14,\, 20 \,\}$$

Então a remoção do primeiro elemento com a eliminação de incompatíveis produz o subproblema

$$A' \;=\; \{\, 17,\, 11,\, 14,\, 20 \,\}$$

Enquanto que apenas a remoção do primeiro elemento produz o subproblema

$$A' \quad = \quad \{\, 17,\ 5,\ 11,\ 7,\ 14,\ 20 \,\}$$

Para levar esse detalhe em consideração, nós observamos que um subproblema $A' = \{a_i, ..., a_n\}$ pode ter elementos incompatíveis eliminados de no máximo n maneiras — uma para cada elemento que aparece antes de a_i.

(Você consegue ver isso?)

Portanto, a execução do algoritmo pode produzir $O(n^2)$ subproblemas diferentes.

E esse é o número de chamadas recursivas diferentes que são realizadas.

A seguir, nós precisamos estimar o tempo de execução de cada chamada recursiva.

É fácil ver que a operação de `Remove-e-Elimina` pode ser realizada em tempo $O(n)$.

Desta vez não há laço, e todo o resto executa em tempo $O(1)$.

Portanto, o algoritmo como um tempo executa em tempo

$$O(n^2) \quad \times \quad O(n) \quad = \quad O(n^3)$$

Ou seja, melhorou ...

Mas, ainda é possível fazer melhor que isso com uma pequena esperteza.

A ideia é não eliminar mais os elementos incompatíveis da sequência.

Ao invés disso, nós acrescentamos um parâmetro adicional `k`, que indica ao algoritmo que ele não deve considerar nenhum elemento menor ou igual a `k` para a solução.

Abaixo nós temos a nova versão do algorimo

```
Procedimento  SC+L-PD2.1  ( A: sequência com n números ; k: inteiro )
 {
    Se ( A está vazio )   Retorna (vazio)

    Se ( a1 <= k )
     {
        A'  <--  Remove (A,a1)           // só remove a1
        Retorna  SC+L-PD2.1 (A',k)
     }

    A1  <--  Remove (A,a1)               // só remove a1
    S1  <--  SC+L-PD2.1 (A1,a1)
    S1  <--  Concatena (a1,S1)
```

```
   A2  <--  Remove (A,a1)                // só remove a1
   S2  <--  SC+L-PD2.1 (A2,k)

   Se ( tamanho(S1) > tamanho(S2) )    Retorna (S1)
   Senão                               Retorna (S2)
}
```

Agora, as chamadas recursivas executam em tempo $O(1)$.

Mas, ainda existem $O(n^2)$ subproblemas.

(Porque?)

Portanto, o algoritmo como um todo executa em tempo $O(n^2)$.

2 O problema da edição

O que é mais parecido com o tomate: o abacaxi ou o abacate?

A resposta é: o abacate.

Veja só:

```
- T O M A T E          - T O M A T E
A B A C A T E          A B A C A X I
```

Por outro lado, o abacate é mais parecido com o abacaxi do que com o tomate:

```
A B A C A X I
A B A C A T E
```

(Apesar de que, na feira, se você pedir um abacate, dificilmente alguém vai te dar um abacaxi, mas você pode acabar ganhando um tomate ...)

E o tamarindo, é mais parecido com o que?

```
T A M A R I N D O - - - -      T A M A R I N D O      - T A M A R I N D O
T - - - - - - - O M A T E      - A B A C A T E -      A B A C A X I - - -
```

O abacaxi!

Hmm, esse negócio de comparar as coisas, sei não ...

A discussão acima ilustra o problema dessa seção de maneira apenas aproximada.

Quer dizer, nós vamos formalizar o problema em termos da diferença, e não a semelhança, entre duas sequências de caracteres

$$\mathtt{X} \;=\; \mathtt{x}_1\ \mathtt{x}_2\ \mathtt{x}_3\ \ldots\ \mathtt{x}_{n-1}\ \mathtt{x}_n \qquad\qquad \mathtt{Y} \;=\; \mathtt{y}_1\ \mathtt{y}_2\ \mathtt{y}_3\ \cdots\ \mathtt{y}_{m-1}\ \mathtt{y}_m$$

E para estimar essa diferença, a ideia é tentar alinhar as duas sequências da melhor maneira possível, permitindo que

- sejam inseridos espaços em branco (-) em qualquer lugar das sequências
- ou que dois símbolos alinhados sejam diferentes

A diferença é definida então como o número total de posições onde os símbolos não são iguais (no melhor alinhamento).

Intuitivamente, esse valor corresponde ao número de operações de inserção, deleção, ou substituição de caracter necessárias para transformar uma sequência na outra — daí o nome *distância de edição*.

Por exemplo, a diferença entre tomate e abacate é igual a 4:

```
- T O M A T E
A B A C A T E
```

Pensando em termos de divisão e conquista, a ideia é tentar decompor o problema em dois subproblemas independentes.

A maneira mais natural de fazer isso consiste em escolher um símbolo de cada sequência e alinhá-los

$$\begin{array}{rrrrr|c|lll} \mathsf{x1} & \mathsf{x2} & \mathsf{x3} & \ldots & \ldots & \mathsf{xj} & \ldots & \mathsf{xn\text{-}1} & \mathsf{xn} \\ \mathsf{y1} & \mathsf{y2} & \mathsf{y3} & \ldots & & \mathsf{yk} & \ldots & \ldots & \mathsf{ym\text{-}1}\ \ \mathsf{ym} \end{array}$$

De acordo com o esquema acima, o próximo passo consiste em resolver os subproblemas

$$A_1 \;:\; \Big(\mathtt{X}[1\,..\,\mathtt{j}-1]\,;\,\mathtt{Y}[1\,..\,\mathtt{k}-1]\Big) \qquad\qquad A_2 \;:\; \Big(\mathtt{X}[\mathtt{j}+1\,..\,\mathtt{n}]\,;\,\mathtt{Y}[\mathtt{k}+1\,..\,\mathtt{m}]\Big)$$

E calcular a diferença da solução como

$$\mathsf{Dif}(S) \;=\; \mathsf{Dif}(S_1) \;+\; \mathsf{Dif}(S_2) \;+\; \mathsf{dif}(\mathtt{x}_\mathtt{j}, \mathtt{y}_\mathtt{k})$$

onde

$$\mathsf{dif}(\mathtt{x}_\mathtt{j}, \mathtt{y}_\mathtt{k}) \;=\; \begin{cases} 0 & \text{, se } \mathtt{x}_\mathtt{j} \;=\; \mathtt{y}_\mathtt{k} \\ 1 & \text{, caso contrário} \end{cases}$$

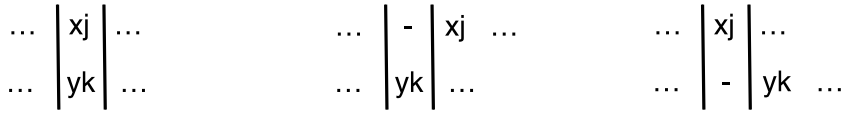
Como usual, isso terá que ser feito para todas as escolhas possíveis de símbolos em `X` e `Y`.

E a solução ótima será dada pela menor diferença entre todas as possibilidades.

Mas, ainda falta um detalhe.

Lembre que nós podemos introduzir espaços em branco (`-`) nas duas sequências, para tentar construir um bom alinhamento.

Isso significa basicamente que existem 3 possibilidades para cada par de símbolos x_j, y_k:



E ao fazer isso nós consideramos todas as possibilidades de alinhamento entre `X` e `Y`.

(Você consegue ver isso?)

Abaixo nós temos o pseudo-código do algoritmo

```
Procedimento  Dist-Ed-PD1.0  ( X[i..s] , Y[l..t] )
 {
    Se ( X está vazio )   Retorna ( comprimento(Y) )
    Se ( Y está vazio )   Retorna ( comprimento(X) )

    Smin  <--  infinito

    Para cada para de símbolos  xj, yk
     {
        S1  <-- Dist-Ed-PD1.0 ( X[i..j-1] , Y[l..k-1] )  +  1
                +  Dist-Ed-PD1.0 ( X[j+1..s], Y[k..l] )

        S2  <-- Dist-Ed-PD1.0 ( X[i..j-1] , Y[l..k-1] )  +  1
                +  Dist-Ed-PD1.0 ( X[j..s], Y[k+1..l] )

        S3  <-- Dist-Ed-PD1.0 ( X[i..j-1] , Y[l..k-1] )
                +  dif (xj,yk)
                +  Dist-Ed-PD1.0 ( X[j..s], Y[k+1..l] )

         S  <--  Min { S1, S2, S3 }

        Se ( S < Smin )   Smin  <--  S
     }
    Retorna (Smin)
 }
```

Análise de complexidade

É fácil determinar o número de subproblemas gerados durante a execução:

- Existem $n \times n$ possibilidades para os índices i, s na sequência `X`
- Existem $n \times n$ possibilidades para os índices l, t na sequência `Y`

Logo, existem $O(n^4)$ subproblemas.

Por outro lado, o laço do algoritmo executa no máximo $O(n^2)$ voltas, e tudo o mais executa em tempo $O(1)$.

Portanto, o tempo total de execução é $O(n^6)$ — argh!

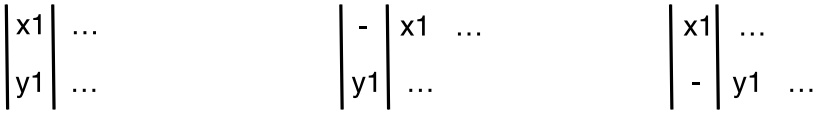
Estratégia alternativa de decomposição

A seguir, nós vamos pensar sobre o problema em termos de algoritmos gulosos.

O raciocínio guloso seria algo assim: "*Tente alinhar o primeiro símbolo de* `X` *com o primeiro símbolo de* `Y` *(ou com um branco), e reduza o problema só um pouquinho.*"

Suponha, portanto, que nós selecionamos os símbolos $\mathtt{x}_1, \mathtt{y}_1$ para começar a construção da solução.

Como sabemos, será preciso analisar 3 casos:



E cada um desses casos irá produzir um novo subproblema.

A ideia, então, é resolver os 3 subproblemas e ver qual deles nos dá a menor diferença.

Abaixo nós temos o pseudo-código do algoritmo

```
Procedimento  Dist-Ed-PD2.0  ( X[i..n] , Y[l..m] )
 {
    Se ( X está vazio )   Retorna ( comprimento(Y) )
    Se ( Y está vazio )   Retorna ( comprimento(X) )

    S1  <--  1  +  Dist-Ed-PD2.0 ( X[j+1..n], Y[l..m] )

    S2  <--  1  +  Dist-Ed-PD2.0 ( X[j..n], Y[l+..m] )

    S3  <--  dif (xi,yl)
                +  Dist-Ed-PD2.0 ( X[j..n], Y[l+..m] )
```

```
    Retorna ( Min { S1, S2, S3 } )
}
```

Análise de complexidade

Note que

- Existem n possibilidades para o índice i na sequência `X`
- Existem n possibilidades para o índice l na sequência `Y`

Logo, existem $O(n^2)$ subproblemas.

Por outro lado, o algoritmo não possui mais laço, e tudo ainda executa em tempo $O(1)$.

Portanto, o tempo total de execução é $O(n^3)$ — agora sim ...