Construção e Análise de Algoritmos

aula 23: Dois problemas sobre partições

1 Introdução

```
Todo mundo tem lá as suas manias, não é mesmo?
Tem uns que nunca pisam nos riscos da calçada.
Outras pessoas só andam se pisarem nos riscos.
Alguns evitam branco e só botam o pé no preto.
Outros contam certinho e pisam igual nos dois.
Há quem apenas começa se for com o pé direito.
Outros tem medo de terminar com o pé esquerdo.
Há muito tempo, no começo da era da Computação
tudo era feito em terminais de texto com letra
de largura fixa, e a cabeça da gente sabe como
é, nunca se assossega para fazer uma coisa só.
O sujeito ia escrevendo e pensando no assunto,
e ia ao mesmo tempo contando as letrinhas para
ver se ele conseguia deixar super arrumadinho.
Dava um trabalho danado, é claro, mas no final
a alegria era grande de ver que nos dois lados
tudo tinha ficado igual, as duas margens muito
bem alinhadinhas, com o mesmo número de letras
e espaços em cada linha, contados com precisão.
Mas, a gente sabe que essas coisas não existem
mais, isso é coisa de antigamente que há muito
tempo ficou para trás, e se alguém ainda tenta
brincar desse tipo de coisa, é porque no final
das contas, está tentando é fazer outra coisa.
```

Bom, esse é o primeiro problema que nós vamos discutir na aula de hoje.

Quer dizer, o problema de introduzir quebras de linha entre as palavras de um texto de modo que cada linha contenha aproximadamente o mesmo número de caracteres.

2 O problema do alinhamento

Na brincadeira acima, na maior parte dos casos, as palavras e mesmo algumas ideias de cada linha foram escolhidas de propósito para produzir um alinhamento perfeito.

Em um problema real, no entanto, o texto já está definido, e tudo o que se pode esperar é que os erros de alinhamento sejam os menores possíveis.

Existem muitas maneiras de medir esse erro.

Nós vamos assumir que cada linha no papel pode conter até C caracteres.

E quando uma linha contém menos do que C caracteres, isso corresponde a um erro de alinhamento.

Além disso, intuitivamente, é melhor ter duas linhas com erro igual a 2, do que uma linha com erro 1 e outra com erro 3.

Para refletir essa ideia, nós penalizamos cada linha com o quadrado do valor do seu erro.

Dessa maneira, a primeira alternativa teria penalidade acumulada igual a $2^2 + 2^2 \;=\; 8$, e a segunda $1^2 + 3^2 \;=\; 10$.

Pronto, agora nós já podemos formalizar o nosso problema

- Dada uma sequência de palavras

  $$p_1,\;\; p_2,\;\; p_3,\;\; \ldots,\;\; p_{n-1},\;\; p_n$$

  o problema consiste em dividir essas palavras em linhas

  $$L_1,\;\; L_2,\;\; \ldots,\;\; L_k$$

  de modo que o erro de alinhamento

  $$\sum_{j=1}^{k} \Big[\; C \;\; - \;\; \big(\sum_{p_i \in L_j} \mathsf{compr}(p_i) \;\; + \;\; |L_j| - 1\big) \;\Big]^2$$

  seja o menor possível

Note que a função de custo acima leva em conta os espaços entre as palavras de uma linha, para que eles não contribuam para o erro.

O esquema abaixo deixa claro que uma solução do problema corresponde a uma partição da sequência de palavras

$$\underbrace{p_1,\;\; p_2,\;\; p_3}_{L_1} \;\Big|\; \underbrace{p_4,\;\; p_5}_{L_2} \;\Big|\; \ldots \quad \ldots \;\Big|\; \underbrace{p_{n-2},\;\; p_{n-1},\;\; p_n}_{L_k}$$

E o problema de otimização consiste em encontrar uma partição com custo mínimo.

A ideia é resolver o problema por programação dinâmica.

E o primeiro passo consiste em tentar quebrar o problema em dois subproblemas independentes, que podem ser resolvidos em separado.

Isso é bem fácil, veja só

$$p_1,\ p_2,\ \ldots,\ p_i \ \Big|\ p_{i+1},\ \ldots,\ p_{n-1},\ p_n$$

Quer dizer, ao assumir que ocorre uma quebra de linha entre as palavras p_i e p_{i+1}, o texto original é dividido em dois textos menores que agora precisam ser alinhados da melhor maneira possível.

E também é fácil de ver que a penalidade dessa solução S é dada pela soma das penalidades das soluções dos dois subproblemas

$$\mathsf{Pen}(S) \ = \ \mathsf{Pen}(S_1) \ + \ \mathsf{Pen}(S_2)$$

A última observação é que não há qualquer garantia de que a introdução de uma quebra de linha entre as palavras p_i e p_{i+1} nos leva até a solução ótima.

Portanto, a ideia é que nós vamos ter que considerar todas possibilidade de lugar para a introdução da quebra de linha.

Abaixo nós temos o algoritmo código que implementa essa ideia (com a memoização omitida).

```
   Procedimento  Alinhamento-PD1.0 ( {pi, ..., pj}: sequência de palavras )
    {
1.     Se ( as palavras  pi,...,pj  cabem em uma linha )
        {
2.        S  <--  {pi,...,pj};    P  <--  Calcula-Penalidade (S)
3.        Retorna (S,P)
        }

4.     Smin  <--  Nulo;    Pmin  <--  infinito

5.     Para  k <-- i  Até  j-1
        {
6.        (S1,P1)  <--  Alinhamento-PD1.0 ( {pi,...,pk} )
7.        (S2,P2)  <--  Alinhamento-PD1.0 ( {pi,...,pk} )

8.        Se ( Pmin > P1 + p2 )
           {
9.            Smin  <--  Combina (S1,S2);    Pmin  <--  P1 + P2
           }
        }
10.    Retorna (Smin,Pmin)
    }
```

Análise de Complexidade

Note que cada chamada recursiva é caracterizada pelos índices i, j da subsequência de palavras $\{p_i, \ldots, p_j\}$ que ela recebe como entrada.

Cada um desses índices pode assumir valores entre 1 e n.

Portanto, o algoritmo realiza $O(n^2)$ chamadas recursivas distintas.

A seguir, nós observamos que, com a exceção da cálculo da penalidade na linha `2`, todas as instruções desse procedimento executam em tempo $O(1)$.

Examinando a expressão da penalidade na definição do problema, nós vemos que ela pode ser calculada em tempo $O(n)$.

Além disso, note que o laço da linha `5` realiza $O(n)$ voltas.

Portanto, cada chamada recursiva executa em tempo $O(n)$.

Finalmente, o tempo de execução do algoritmo como um todo é dado por

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

2.1 Uma estratégia de decomposição mais esperta

Como vimos na aula passada, a ideia de decompor um problema em dois subproblemas menores nem sempre nos dá o algoritmo mais eficiente.

A estratégia alternativa consiste na ideia de decompor o problema em apenas um subproblema, um pouquinho menor que o original.

No caso do nosso problema, isso corresponde a escolher o lugar onde a primeira quebra de linha será introduzida.

Por exemplo,

$$p_1,\ p_2,\ p_3 \ \Big|\ p_4,\ \ldots,\ p_{n-1},\ p_n$$

Antes mesmo de escrever o algoritmo baseado nessa ideia e analisar a sua complexidade, nós já podemos apreciar as vantagens dessa abordagem:

- Em primeiro lugar, o número de possibilidades de quebra a ser considerado tende a ser bem menor, pois uma linha só pode conter no máximo C caracteres.
- Em segundo lugar, os subproblemas que são produzidos tem o formato mais simples $\{p_i, \ldots, p_n\}$, e existe um número menor deles.

Essas observações tem um impacto direto no número total de chamadas recursivas que são realizadas, e no tempo de execução de cada chamada.

Vejamos.

```
   Procedimento  Alinhamento-PD2.0 ( {pi, ..., pn}: sequência de palavras )
    {
1.     Se ( as palavras  pi,...,pn  cabem em uma linha )
        {
2.        S  <--  {pi,...,pn};    P  <--  Calcula-Penalidade (S)
3.        Retorna (S,P)
        }

4.     Smin  <--  Nulo;    Pmin  <--  infinito

5.     k  <--  i
6.     Enquanto ( {pi,...,pk}  cabem em uma linha )
        {
7.        S1  <--  {pi,...,pk};    P1  <--  Calcula-Penalidade (S1)

8.        (S2,P2)  <--  Alinhamento-PD2.0 ( {pk+1,...,pn} )

9.        Se ( Pmin > P1 + p2 )
           {
10.           Smin  <--  Combina (S1,S2);    Pmin  <--  P1 + P2
           }

11.       k++
        }
12.    Retorna (Smin,Pmin)
    }
```

Como cada chamada recursiva é caracterizada por um subproblema $\{\mathtt{p_i}, ..., \mathtt{p_n}\}$, e só existem n possibilidades para o valor do índice i, nós concluímos que o algoritmo só realiza $O(n)$ chamadas recursivas.

Além disso, note que

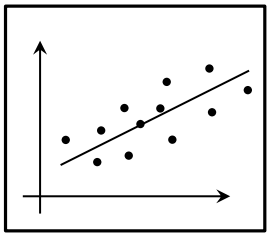
- A chamada ao procedimento `Calcula-Penalidade`, na linha 2, leva tempo $O(C)$ — aqui nós estamos fazendo uma estimativa mais precisa do que na análise do algoritmo anterior.
- O laço da linha 6 realiza $O(C)$ voltas, pois cada palavra p_k possui ao menos 1 caracter.
- Por outro lado, as chamadas a `Calcula-Penalidade` na linha 7 podem ser otimizadas para executar em tempo $O(1)$, pois a cada volta do laço apenas uma nova palavra é adicionada à partição S_1.
- Dessa maneira, todas as instruções dentro laço executam em tempo $O(1)$.

Portanto, o tempo de execução do algoritmo é de

$$O(n) \quad \times \quad O(C) \quad = \quad O(nC)$$

3 O método dos mínimos quadrados segmentado

Um problema muito comum na área de análise de dados consiste em aproximar uma coleção de pontos no plano por uma linha reta



Quer dizer, quando a aproximação é boa, nós podemos imaginar que os pontos são gerados por um processo que realmente se comporta de acordo com a equação

$$y \quad = \quad a \cdot x \; + \; b$$

e que algum ruído aleatório perturba o valor de y, fazendos com que os pontos se espalhem.

Mas, o que é uma boa aproximação?

E como nós estimamos o erro dessa aproximação?

A intuição aqui é a mesma que vimos no problema do alinhamento.

Quer dizer, o erro de aproximação associado a um certo ponto (x_i, y_i) é definido como

$$\left| \, y_i \; - \; (a \cdot x_i + b) \, \right|$$

ou, a diferença entre o valor real y_i e o valor de y estimado pela aproximação.

E nós preferimos ter uma aproximação relativamente boa para dois pontos, do que ter uma aproximação que é muito boa para um deles e ruim para o outro.

Como já vimos, isso pode ser obtido considerando o quadrado do erro de cada ponto e somando para todos os pontos

$$\mathsf{Erro} \quad = \quad \sum_i \left| \, y_i \; - \; (a \cdot x_i + b) \, \right|^2$$

Agora que temos uma definição precisa para o erro de aproximação, nós podemos dizer que uma boa aproximação é aquela que minimiza esse erro.

Felizmente, esse problema tem uma solução conhecida.

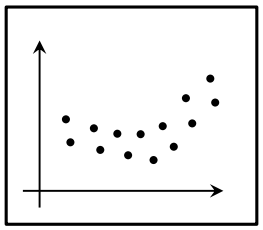
Quer dizer, escolhendo os valores de a e b como*

$$a = \frac{n \cdot \sum_i x_i y_i - \left(\sum_i x_i\right)\left(\sum_i y_i\right)}{n \cdot \sum_i x_i^2 - \left(\sum_i x_i\right)^2} \qquad b = \frac{\sum_i y_i - \sum_i x_i}{n}$$

nós obtemos a melhor aproximação linear possível para a coleção de pontos (de acordo com a definição de erro acima).

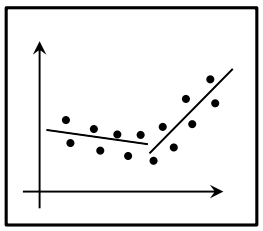
Legal.

Mas, o que nós podemos fazer quando a nossa coleção de pontos tem o seguinte formato



Nesse caso, é fácil ver que mesmo a aproximação linear ótima não nos dá uma boa aproximação para todos os pontos.

Mas, com duas retas a coisa funciona bem.

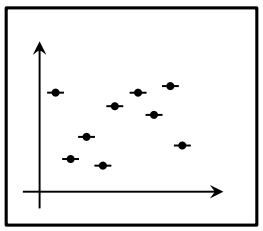


E, nos casos em que duas retas não funcionam, nós podemos utilizar 3 retas.

Mas ...

... não se pode ir tão longe dessa maneira.

Veja só



*para a nossa discussão de hoje, não importa muito como se chega nessas equações ...

Quer dizer, existem basicamente dois problemas aqui:

1. isso não é realmente uma aproximação, mas uma descrição alternativa da coleção de pontos
2. ao descrever os pontos assim individualmente, nós não estamos descrevendo apenas os pontos mas os ruídos também

A segunda observação indica que o objetivo da aproximação não é só obter uma descrição sucinta dos pontos, mas também tentar eliminar parte do ruído associado a eles.

Mas isso só funciona se o número de linhas utilizado na aproximação não é muito grande — ou, equivalentemente, se cada linha tem um grande número de pontos associados a ela.

Nós estamos, portanto, diante de um dilema

- por um lado, nós queremos usar mais linhas para melhorar a qualidade da aproximação
- por outro lado, nós não queremos usar linhas demais para não comprometer o fator de redução de ruído

Qual é o número ideal de linhas em cada caso?

Bom, a ideia aqui será deixar a programação dinâmica decidir essa questão para nós.

Para fazer isso, nós vamos utilizar dois componentes de custo

- nós vamos associar um custo C a cada linha utilizada na solução
- e nós vamos somar os erros de aproximação associados a todas essas linhas

Intuitivamente, quando nós aumentamos o número de linhas na solução, o primeiro componente de custo aumenta e o segundo componente diminui, e vice-versa.

Mais precisamente, dada uma coleção de pontos

$$P \quad = \quad \{p_1,\ p_2,\ p_3,\ \ldots,\ p_{n-1},\ p_n\}$$

onde $p_i = (x_i, y_i)$, e nós assumimos que os pontos estão ordenados pela coordenada x, uma solução consiste em uma partição dos pontos em uma coleção de segmentos

$$S \quad = \quad \{S_1,\ S_2,\ \ldots,\ S_m\}$$

onde os segmentos correspondem a faixas contíguas de pontos da coleção P sem interseção (e.g., $S_i = \{p_j, p_{j+1}, \ldots, p_k\}$).

Para cada solução S, o seu custo é definido por

$$\mathsf{custo}(S) \quad = \quad |S| \cdot C \; + \; \sum_i \mathsf{Erro}(S_i)$$

onde $|S|$ é o número de segmentos na solução S.

Agora que tudo já está bem definido, o próximo passo é construir um algoritmo de programação dinâmica para encontrar uma solução ótima para o problema.

Como usual, existem basicamente duas maneiras de organizar as ideias.

A primeira delas consiste em pensar como na divisão e conquista, e verificar se é possível decompor o problema em subproblemas menores que podem ser resolvidos de maneira independente.

Observando que uma solução qualquer é uma partição dos pontos da coleção P

$$p_1,\ p_2,\ p_3 \ \mid\ p_4,\ p_5 \ \mid\ \ldots\ \ldots \ \mid\ p_{n-2},\ p_{n-1},\ p_n$$

nós podemos imaginar que uma das divisões dessa partição pode estar passando entre os pontos p_i e p_{i+1}

$$p_1,\ p_2,\ \ldots\ \ldots\ p_i \ \mid\ p_{i+1},\ \ldots\ \ldots\ p_n$$

de modo que os pontos $p_1, \ldots, p_i$ serão particionados entre si, e os pontos $p_{i+1}, \ldots, p_n$ serão particionados entre si.

Em outras palavras, aqui nós já temos uma decomposição do problema em dois subproblemas independentes

$$\underbrace{p_1,\ p_2,\ \ldots\ \ldots\ p_i}_{P_1} \ \mid\ \underbrace{p_{i+1},\ \ldots\ \ldots\ p_n}_{P_2}$$

e a solução do problema P é obtida como a união das soluções dos subproblemas P_1 e P_2

$$S \quad = \quad S_1 \ \cup \ S_2$$

(lembre que S_1 e S_2 são coleções de segmentos)

Como nós já sabemos muito bem, não há garantia de que a decisão inicial de separar p_i e p_{i+1} nos leve dê uma solução ótima.

Mas nós também já sabemos como resolver isso: basta examinar todas as possibilidades para essa divisão.

Para completar a definição do algoritmo, ainda falta levar um pequeno detalhe em conta.

Quer dizer, nem sempre particionar a coleção de pontos é a melhor ideia.

Às vezes, a melhor solução é aproximar a coleção de pontos com apenas uma linha.

Portanto, além de calcular as diversas soluções S_i obtidas separando os pares de pontos p_i, p_{i+1}, nós também calculamos a solução S_0 que consiste em aproximar a coleção inteira de pontos com apenas uma linha.

A solução final do problema será solução de menor custo entre todas elas.

Abaixo nós temos o algoritmo de programação dinâmica baseado nessa ideia.

```
   Procedimento  MQS-PD1.0 ( P[i..j]: coleção de pontos )
    {
1.     S  <--  P[i..j]
2.     E  <--  C  +  Erro (S)

3.     Para  k <-- i  Até  j-1
        {
4.         (S1,E1)  <--  MQS-PD-1.0 ( P[i..k] )
5.         (S2,E2)  <--  MQS-PD-1.0 ( P[k+1..j] )

6.         Se ( E > E1 + E2 )
            {
7.             S  <--  S1 + S2;    E  <--  E1 + E2
            }
        }
8.     Retorna (S,E)
    }
```

Para estimar o tempo de execução do algoritmo, basta observar que

- cada chamada recursiva realiza um laço que dá no máximo n voltas,
  e todas as suas instruções executam em tempo $O(1)$
- cada chamada recursiva recebe um subproblema `P[i..j]`, e existem
  no máximo n^2 subproblemas desse tipo (um para cada par de valores para i, j)

Portanto, a complexidade do algoritmo é

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

Estratégia de decomposição alternativa

A segunda ideia consiste em pensar como nos algoritmos gulosos, e tentar reduzir o problema pelas beiradas.

Nesse caso, ao invés de pensar em uma segmentação qualquer que divide o problema no meio,

nós consideramos as possibilidades para a localização da primeira divisão

$$p_1 \mid p_2,\ p_3,\ p_4,\ \ldots\ p_{n-1},\ p_n$$

$$p_1,\ p_2 \mid p_3,\ p_4,\ \ldots\ p_{n-1},\ p_n$$

$$p_1,\ p_2,\ p_3 \mid p_4,\ \ldots\ p_{n-1},\ p_n$$

$$\ldots$$

Em todos esses casos, uma vez definido o primeiro segmento da partição, o que resta é um subproblema da forma `P[j..n]` que pode ser resolvido de maneira independente.

E, como na solução anterior, também é preciso levar em conta o caso em que toda a coleção de pontos é aproximada por apenas uma linha.

Abaixo nós temos o algoritmo de programação dinâmica baseado nessa ideia.

```
   Procedimento  MQS-PD2.0 ( P[j..n]: coleção de pontos )
    {
1.     S  <--  P[j..n]
2.     E  <--  C  +  Erro (S)

3.     Para  k <-- j  Até  n-1
        {
4.              S1  <--  P[j..k]
5.              E1  <--  C  +  Erro (S1)
6.         (S2,E2)  <--  MQS-PD-1.0 ( P[k+1..j] )

7.         Se ( E > E1 + E2 )
            {
8.             S  <--  S1 + S2;    E  <--  E1 + E2
            }
        }
9.     Retorna (S,E)
    }
```

Para estimar o tempo de execução do algoritmo, basta observar que

- cada chamada recursiva realiza um laço que dá no máximo n voltas, e todas as suas instruções executam em tempo $O(1)$
- cada chamada recursiva recebe um subproblema `P[j..n]`, e existem no máximo n subproblemas desse tipo (um para valor de j)

Portanto, a complexidade do algoritmo é

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