Construção e Análise de Algoritmos

aula 10: Alguns problemas clássicos

1 Introdução

Todos aprendem na escola a fazer multiplicação da seguinte maneira

$$\begin{array}{r} 371 \\ \times\ 58 \\ \hline 2968 \\ 1855 \\ \hline 21518 \end{array}$$

Quer dizer,

- `o primeiro número (todos os seus dígitos) é multiplicado por cada dígito do segundo`
- `e depois os resultados (deslocados uma casa cada um) são somados.`

— (*esse é o algoritmo ...*)

Daí, assumindo que os números tem n dígitos, não é difícil ver que a coisa leva tempo $O(n^2)$.

Todo mundo também aprende na escola a multiplicar matrizes assim

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \times \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} (1\cdot 5 + 2\cdot 7) & (1\cdot 6 + 2\cdot 8) \\ (3\cdot 5 + 4\cdot 7) & (3\cdot 6 + 4\cdot 8) \end{bmatrix}$$

Quer dizer,

- `cada linha da primeira matriz é multiplicada por cada coluna da segunda matriz`

Daí, assumindo que as matrizes tem dimensão $n \times n$, não é difícil ver que a coisa leva tempo $O(n^3)$.

À primeira vista, não dá para fazer essas multiplicações mais rápido do que isso.

Mas hoje, nós vamos ver como isso pode ser feito.

2 Estratégia de decomposição

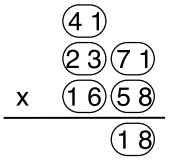
Considere primeiramente o problema da multiplicação de inteiros.

A primeira observação é que, ao invés de trabalhar com os dígitos individualmente, nós podemos, por exemplo, trabalhar com pares de dígitos

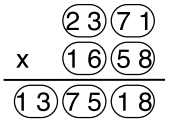
$$\begin{array}{r} (23)(71) \\ \times\ (16)(58) \\ \hline \end{array}$$

A ideia aqui é começar multiplicando o primeiro número (formado pelos pares 23 e 71) pelo par 58 — isso é a mesma coisa que trabalhar na base 100 (a aritmética da centopéia ...).

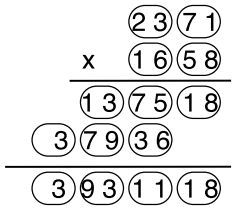
Ao multiplicar 58 por 71, nós obtemos 18 e "vai 41":[1]



A seguir, multiplicando 58 por 23, e somando 41, nós obtemos



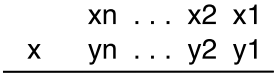
Finalmente, repetindo a mesma operação para o par 16 e somando os resultados, nós obtemos a resposta



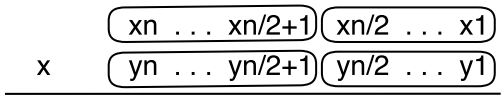
Legal!

Agora nós estamos prontos para pensar sobre números bem grandes.

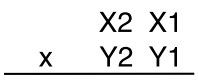
Considere a multiplicação abaixo, envolvendo dois números X e Y com n dígitos cada um



A ideia dessa vez é trabalhar com grupos de $n/2$ dígitos, dividindo cada número exatamente no meio



Denotando as duas partes de X e Y por X_2, X_1 e $Y2, Y1$, respectivamente, a coisa fica assim



E agora, basta fazer as contas.

Observando que

$$X = X_2 \cdot 10^{n/2} + X_1 \qquad Y = Y_2 \cdot 10^{n/2} + Y_1$$

as contas podem ser feitas da seguinte maneira

$$\begin{aligned} X \cdot Y &= \left(X_2 \cdot 10^{n/2} + X_1\right) \times \left(Y_2 \cdot 10^{n/2} + Y_1\right) \\ &= X_2Y_2 \cdot 10^n + X_2Y_1 \cdot 10^{n/2} + X_1Y_2 \cdot 10^{n/2} + X_1Y_1 \end{aligned}$$

[1] Essa é a aritmética da centopéia ...

3 Uma estratégia de conquista esperta

Infelizmente, a estratégia de decomposição que nós acabamos de ver ainda não produz um resultado interessante.

Vejamos.

Nós já sabemos que a multiplicação de X e Y dígito a dígito leva tempo n^2.

De maneira análoga, as multiplicações envolvendo X_i,Y_j (que possuem $n/2$ dígitos cada um) levam tempo $\frac{n}{2} \cdot \frac{n}{2} = \frac{n^2}{4}$.

Mas, como são realizadas 4 multiplicações desse tipo, o tempo total é de

$$4 \cdot \frac{n^2}{4} \quad = \quad n^2$$

Ou seja, nós não ganhamos nada até aqui.

A observação chave, a seguir, é que a multiplicação de X e Y também pode ser escrita como

$$X \cdot Y \quad = \quad X_2Y_2 \cdot 10^n \quad + \quad \big(X_2Y_1 \quad + \quad X_1Y_2\big) \cdot 10^{n/2} \quad + \quad X_1Y_1$$

e aqui nós vemos que nós não precisamos dos resultados intermediários X_2Y_1 e X_1Y_2 separadamente, mas apenas da sua soma.

E agora, tudo o que é preciso é uma pequena esperteza ...

Veja o que acontece quando nós deixamos as potências de 10 de lado e multiplicamos

$$\big(X_2 + X_1\big) \cdot \big(Y_2 + Y_1\big) \quad = \quad X_2Y_2 \quad + \quad \big(X_2Y_1 \quad + \quad X_1Y_2\big) \quad + \quad X_1Y_1$$

Ou seja, o termo que nos interessa apareceu de novo.

E a observação importante aqui é que, se as somas do lado esquerdo são realizadas primeiro, então a coisa toda leva tempo $\frac{n}{2} + \frac{n}{2} + \frac{n^2}{4}$.

O único problema é que essa multiplicação nos dá o termo que nós desejamos misturado com X_2Y_2 e X_1Y_1.

Mas, isso é um problema fácil de resolver

- basta calcular os termos X_2Y_2 e X_1Y_1 em separado
- e depois fazer a subtração

$$\big(X_2Y_1 \quad + \quad X_1Y_2\big) \quad = \quad X_2Y_2 \quad + \quad \big(X_2 + X_1\big) \cdot \big(Y_2 + Y_1\big) \quad + \quad X_1Y_1$$

Ora, mas os termos X_2Y_2 e X_1Y_1 já eram termos que eram calculados na solução original.

Portanto, não há perda de tempo (realmente) aqui.

Em resumo, a esperteza consiste em calcular apenas 3 substituições:

- X_2Y_2

- X_1Y_1
- $X_2Y_2 \;+\; \big(X_2 + X_1\big) \cdot \big(Y_2 + Y_1\big) \;+\; X_1Y_1$

(O procedimento também envolve algumas somas e subtrações, mas a sua contribuição para o tempo total não é relevante.)

Finalmente, uma vez que esses 3 termos foram calculados, a solução é obtida através de

$$X \cdot Y \;=\; X_2Y_2 \cdot 10^n \;+\; \big(X_2Y_1 \;+\; X_1Y_2\big) \cdot 10^{n/2} \;+\; X_1Y_1$$

Pseudo-código e análise de complexidade

Uma vez que as estratégias de divisão e conquista foram definidas, é imediato escrever o algoritmo que implementa essa solução:

```
Procedimento  Mult-intDC ( X,Y : n dígitos )
 {
     Se  ( X e Y são pequenos )  Retorna ( X * Y )

     (X2,X1)  <--  Quebra (X)
     (Y2,Y1)  <--  Quebra (Y)

          C1  <--  Mult-intDC (X2,Y2)
          C2  <--  Mult-intDC (X1,Y1)
          C3  <--  Mult-intDC (X2+X1,Y2+Y1)  -  C1  -  C2

           R  <--  C1 * 10^n  +  C3 * 10^n/2  +  C2
 }
```

Examinando o pseudo-código, nós obtemos a seguinte equação de recorrência para o tempo de execução do algoritmo

$$T(n) \;=\; 3 \cdot T(n/2) \;+\; O(n)$$

que possui solução

$$T(n) \;=\; \Theta\big(n^{\log_2 3}\big) \;=\; \Theta\big(n^{1{,}585}\big)$$

4 Multiplicação de matrizes eficiente

As ideias que nós acabamos de ver podem ser aplicadas de maneira semelhante ao problema da multiplicação de matrizes.

Considere duas matrizes M, N com dimensões $ntimesn$, particionadas em 4 blocos do mesmo tamanho:

$$\mathsf{M} \;=\; \begin{bmatrix} A & B \\ C & D \end{bmatrix} \qquad\qquad \mathsf{N} \;=\; \begin{bmatrix} E & F \\ G & H \end{bmatrix}$$

Então, a multiplicação de M e N pode ser expressa como

$$\mathsf{M} \cdot \mathsf{N} \;=\; \begin{bmatrix} (AE + BG) & (AF + BH) \\ (CE + DG) & (CF + DH) \end{bmatrix}$$

Como no caso da multiplicação de inteiros, a solução não requer que os produtos AE, BG, ..., DH sejam calculados individualmente.

Tudo o que precisamos são as 4 somas acima.

Daí que, um dia em que V. Strassen não tinha coisa melhor para fazer, ele ficou brincando com isso.

E descobriu que as somas podem ser calculadas realizando apenas 7 multiplicações.

A ideia é a seguinte.

1. primeiro, a gente calcula os seguintes componentes

$$\begin{aligned}
\mathsf{C}_1 &= \big(A + D\big) \cdot \big(E + H\big) \\
\mathsf{C}_2 &= \big(C + D\big) \cdot E \\
\mathsf{C}_3 &= A \cdot \big(F - H\big) \\
\mathsf{C}_4 &= D \cdot \big(G - E\big) \\
\mathsf{C}_5 &= \big(A + B\big) \cdot H \\
\mathsf{C}_6 &= \big(C - A\big) \cdot \big(E + F\big) \\
\mathsf{C}_7 &= \big(B - D\big) \cdot \big(G + H\big)
\end{aligned}$$

2. e depois, os somas são calculadas assim

$$\begin{aligned}
AE + BG &= \mathsf{C}_1 + \mathsf{C}_4 - \mathsf{C}_5 + \mathsf{C}_7 \\
AF + BH &= \mathsf{C}_3 + \mathsf{C}_5 \\
CE + DG &= \mathsf{C}_2 + \mathsf{C}_4 \\
CF + DH &= \mathsf{C}_1 - \mathsf{C}_2 + \mathsf{C}_3 + \mathsf{C}_6
\end{aligned}$$

Dessa maneira, nós obtemos um algoritmo de divisão e conquista para a multiplicação de matrizes com tempo de execução descrito pela recorrência

$$T(n) \;=\; 7 \cdot T(n/2) \;+\; O(n^2)$$

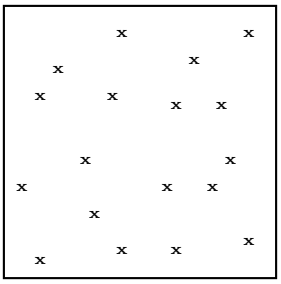
que possui solução

$$T(n) \;=\; \Theta\big(n^{\log_2 7}\big) \;=\; \Theta\big(n^{2{,}8}\big)$$

Seguindo o exemplo de Strassen, outros descobriram decomposições mais sofisticadas, e hoje já se conhecem algoritmos que realizam a multiplicação de matrizes em tempo $O(n^{2{,}37})$.

5 O problema do par de pontos mais próximos

Imagine que nós temos uma coleção de pontos em uma região do plano



Ou melhor, os pontos são dados na forma de uma lista de coordenadas (x, y).

Por exemplo,

L

(8,2)	(1,2)	(5,3)	(7,3)	(2,5)	(3,3)	(4,8)	(3,7)	(1,8)	(5,6)

A tarefa consiste em

→ *Identificar o par de pontos que tem a menor distância entre si*

Como é que a gente faz isso?

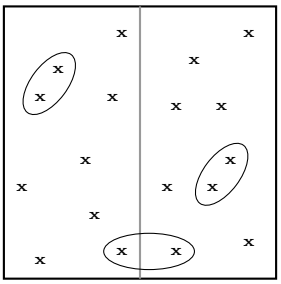
Bom, é fácil escrever um programa que calcula as distâncias entre todos os pares de pontos.

Mas, esse algoritmo é uma porcaria ...

Então, nós vamos aplicar o método da divisão e conquista.

Quer dizer, a ideia é a seguinte

1. primeiro a gente divide a região no meio
2. daí, a gente encontra o par de pontos mais próximos do lado esquerdo
3. depois, encontra o par de pontos mais próximos do lado direito
4. e depois, a gente encontra a melhor solução envolvendo um ponto de cada lado
5. finalmente, a gente escolhe a melhor solução das 3



Abaixo nós temos o código do algoritmo

```
func  Par+Proximo (L, inicio, fim)
 {
     Se ( fim = inicio + 1 )    Retorna ( dist(inicio,fim) )

   meio  <--  (inicio + fim) / 2

     D1  <--  Par+Proximo (L, inicio, meio)

     D2  <--  Par+Proximo (L, meio+1, fim)

     D3  <--  Sol_Fronteira (L, inicio, meio, fim)

     Retorna ( Min { D1, D2, D3 } )
 }
```

Análise e reconstrução

Mas, do jeito que está, o algoritmo ainda é uma porcaria.

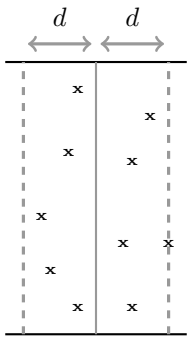
Porque só a função `Sol_Fronteira()` já leva tempo $O(n^2)$ — (*porque?*)

Então, nós vamos precisar de alguma esperteza para consertar isso.

E a esperteza consiste em notar que nós não precisamos examinar todos os pares de pontos (um de cada lado) para encontrar a melhor solução na fronteira.

Quer dizer, imagine que nós já temos uma solução de valor d em um dos dois lados.

Então, a gente só precisa examinar os pontos que estão a uma distância d da fronteira



— (*porque?*)

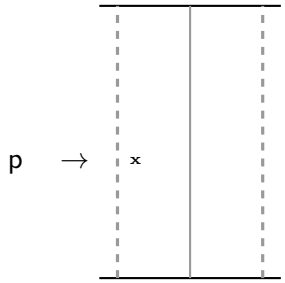
Mas, isso ainda não ajuda muito.

Porque, em princípio, todos os pontos podem estar dentro dessa faixa.

Então agora, nós precisamos de uma segunda esperteza.

Para chegar lá, a gente foca a atenção em um desses pontos[2]

[2]dar um nome é uma forma de focar a atenção



E daí pergunta

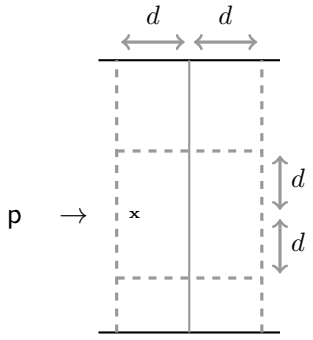
- *Quantos pontos podem estar perto de* p *?*

Quer dizer, se a gente já sabe que um ponto está a distância ao menos d de p, então a gente não precisa calcular a sua distância para p.

Mas, quando é que a gente sabe isso?

Bom, se a diferença entre as coordenadas y dos dois pontos é ao menos d, então a distância entre eles é ao menos d — (*porque?*)

Daí que, agora a gente pode concentrar a atenção no quadradinho



À primeira vista, isso também não ajuda muito.

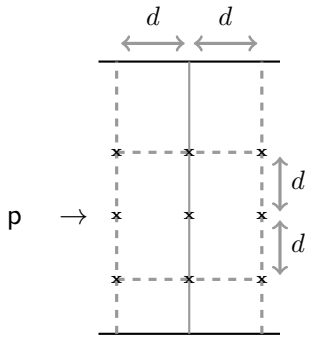
Porque, em princípio, todos os pontos podem estar dentro desse quadradinho.

Mas, isso não é verdade.

Quer dizer, a gente já sabe que a menor distância entre dois pontos do mesmo lado é d.

Logo, não pode haver dois pontos do mesmo lado do quadradinho com distância menor do que d.

No pior caso, a gente teria algo assim



Quer dizer, é uma quantidade $O(1)$ de pontos — (*na vizinhança de* p)

Logo, a inspeção de cada ponto na fronteira leva tempo $O(1)$.

Logo, a função `Sol_Fronteira()` pode executar em tempo $O(n)$.

Implementação e análise

Legal.

Mas agora, é preciso pensar nas questões práticas.

Quer dizer, as questões práticas são duas

1. Como é que a gente particiona a coleção de pontos no meio? — (*para fazer as chamadas recursivas*)
2. Como é que a gente localiza os pontos próximos de p na fronteira? — (*para calcular a melhor solução de fronteira*)

Bom, a resposta para as duas questões é a mesma

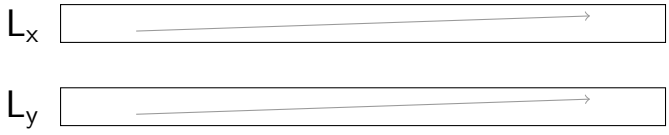
→ *mantendo a lista de pontos ordenada*

Só que, para a primeira questão, nós precisamos ordernar os pontos pela coordenada x.

E para a segunda questão, nós precisamos ordenar os pontos pela coordenada y.

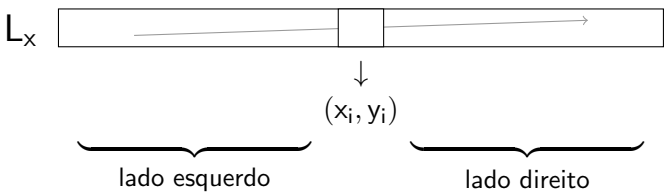
Mas, a boa notícia é que nós só precisamos fazer isso uma vez.

Quer dizer, a gente vai fazer essas ordenações antes de iniciar o algoritmo, e armazenar os resultados em duas listas auxiliares

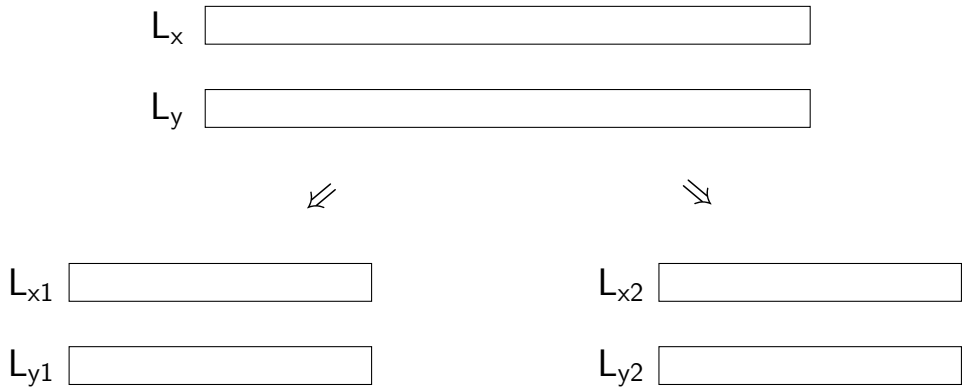


Agora, a divisão da região no meio é fácil.

Quer dizer, basta tomar o elemento do meio em L_x como referêncial



No momento da chamada recursiva, a gente passa não apenas a faixa de pontos corresponde em L_x, mas os mesmos pontos ordenados pela coordenada y — (*filtrados da lista* L_y)



Dessa maneira, cada chamada recursiva começa exatamente da mesma maneira que a chamada anterior.

E a boa notícia é que tudo isso leva tempo $O(n)$.

Certo.

Agora é preciso examinar o cálculo da solução de fronteira.

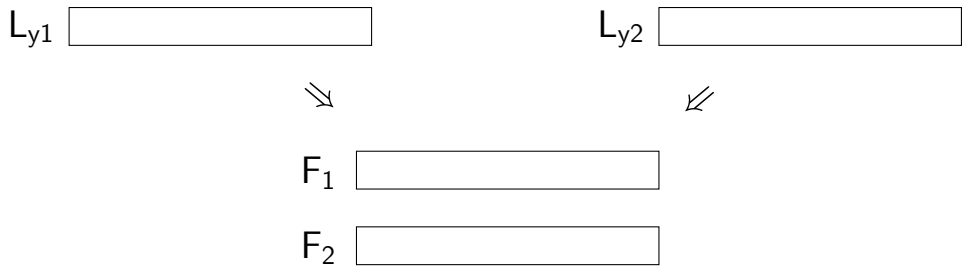
Para isso, é preciso calcular a fronteira.

E para isso, a gente calcula

```
d  <--  Min { D1, D2 }
```

Daí, a gente seleciona em L_{x1} e L_{x2} os pontos que estão à distância no máximo d da fronteira

E filtra esses pontos em L_{y1} e L_{y2} — (*ordenados pela coordenada* y)



Agora é fácil.

Quer dizer, basta fazer a intercalação de F_1 e F_2 — (*pela coordenada y*)

E percorrer a lista resultante, calculando a distância entre cada ponto e os 7 pontos seguintes na lista.

A boa notícia é que tudo isso também leva tempo $O(n)$.

E agora, a gente pode perguntar

- *Qual é o tempo de execução do algoritmo como um todo ?*

Bom, as ordenações antes do algoritmo começar levam tempo

$$O(n \log n)$$

E o algoritmo aplica o método da divisão e conquista

1. dividindo o problema na metade, resolvendo cada metade recursivamente
2. e levando tempo $O(n)$ nas etapas de divisão e conquista

Esse é o mesmo padrão dos algoritmos nos exemplos (a) e (b).

E nós já sabemos que isso leva tempo

$$O(n \log n)$$

Logo, o tempo total do algoritmo é

$$O(n \log n) \quad + \quad O(n \log n) \quad = \quad O(n \log n)$$