Estruturas de Dados

aula 04: A busca binária

1 Introdução

Quase todos os algoritmos que nós vimos até agora são muito, muito ruins ...

Quer dizer, mesmo um algoritmo que executa em tempo $O(n)$ pode ser uma porcaria.

Porque a lista que a gente tem na memória pode ser muito, muito grande.

E daí, procurar um elemento nessa lista vai levar um bom tempo.

Mas o pior mesmo é que, frequentemente, nós queremos procurar um monte de elementos na lista — (*ou modificar, ou remover, ou inserir*)

E daí, se a lista for muito, muito grande, a coisa fica totalmente inviável.

Hoje nós vamos começar a resolver esse problema.

2 A busca pula-pula

A gente começa com a seguinte pergunta

- *Será que no pior caso a busca tem que levar tempo $O(n)$?*

Bom, se a lista está desordenada a resposta é Sim.

Porque no pior caso, a gente tem que olhar todos os elementos para ter certeza se o elemento está lá ou não.

Mas, quando a lista está ordenada a gente não precisa olhar todos os elementos.

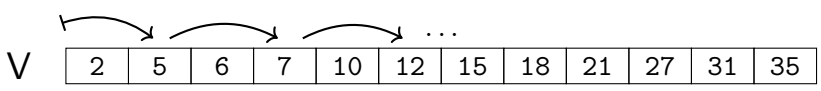
Porque?

Quer dizer, como?

Bom, existem várias maneiras de fazer isso.

E a gente vai começar com a mais simples delas.

Quer dizer, a ideia da busca pula-pula é ir pulando os elementos de 2 em 2, em busca do número x



E daí, podem acontecer várias coisas

1. Se o x está em uma posição par, então a gente cai em cima dele, e ele é encontrado.
2. Se o x está em uma posição ímpar, então a gente passa por cima dele — (*encontrando um elemento maior que x, ou caindo fora da lista*)

   Daí, a gente volta uma posição para encontrá-lo.

3. Se o x está não está na lista, então a gente passa do ponto — (*como no caso 2*)
   E daí, volta uma posição para descobrir que ele não está lá.

A coisa fica assim

```
i = 2
Enquanto ( i ≤ N   E   V[i] < x )                  // pula até não poder mais
    i = i + 2

                                                   // e dai, vê o que aconteceu:
Se ( i > N )                                       // se passou do fim
    Se ( V[N] == x )      Imprime (‘Achei’)
    Senão                 Imprime (‘Não está lá’)
Senão Se ( V[i] == x )    Imprime (‘Achei’)        // se caiu em cima
Senão Se ( V[i-1] == x )  Imprime (‘Achei’)        // se passou por cima
Senão                     Imprime (‘Não está lá’)
```

Análise

Bom, do ponto de vista da notação O, o algoritmo executa em tempo $O(n)$.

Mas desse ponto de vista, a gente não vê o que está acontecendo.

Quer dizer, o laço da primeira etapa só passa por $n/2$ elementos.

E na segunda etapa, a gente só olha mais um ou dois.

Então, esse algoritmo já é duas vezes mais rápido do que a busca na lista desordenada — (*no pior caso*)

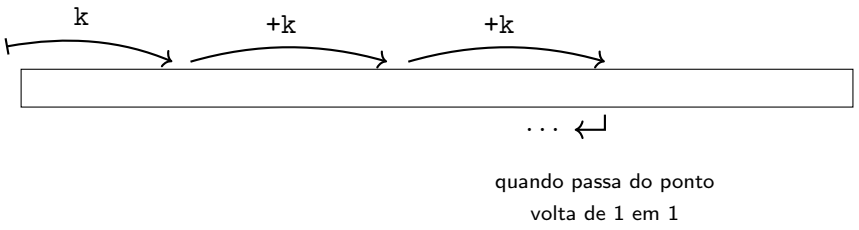
Legal.

Mas, a gente ainda pode fazer melhor.

Quer dizer, a ideia natural é aumentar o tamanho do pulo

- com pulos de tamanho 3, o tempo cai para $n/3$
- com pulos de tamanho 4, o tempo cai para $n/4$
- e por aí vai ...

Para entender o que está acontecendo, a gente considera o algoritmo que dá pulos de tamanho `k`.



quando passa do ponto
volta de 1 em 1

A coisa fica assim

```
i = k
Enquanto ( i ≤ N   E   V[i] < x )       // pula até não poder mais
    i = i + k

Se ( i > N )   i = N                    // volta p/ ver o que aconteceu
achei = 0
Para j = i Até i - k
    Se ( V[i] == x )
        achei = 1;   Quebra
Se ( achei == 1 )     Imprime ('Achei')
Senão                 Imprime ('Não está lá')
```

Análise

Bom, é fácil ver que a primeira etapa examina no máximo

$$\frac{n}{k} \quad \text{elementos}$$

E que a segunda etapa examina no máximo

$$k \quad \text{elementos}$$

Logo, essa busca leva tempo

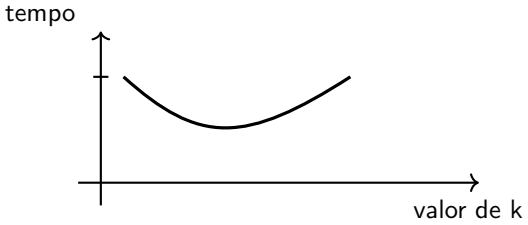
$$\frac{n}{k} \quad + \quad k$$

E agora, a gente pode perguntar

- *Qual é o valor de k que minimiza essa expressão ?*

Bom,

- quando o valor de k cresce, o primeiro termo diminui e o segundo termo aumenta
- quando o valor de k decresce, o primeiro termo aumenta e o segundo termo diminui



Logo, o valor ótimo de k é aquele que faz com que os dois termos fiquem iguais.

$$\frac{n}{k} \quad = \quad k$$

O que nos dá

$$k \quad = \quad \sqrt{n}$$

Quer dizer, a busca pula-pula tem o seu melhor desempenho com pulos de tamanho $\sqrt{n}$.

Com essa configuração, a busca leva tempo

$$\frac{n}{\sqrt{n}} \quad + \quad \sqrt{n} \quad = \quad O(\sqrt{n})$$

Concretamente, em uma lista com 1.000.000 elementos, basta examinar 1000 deles (aprox.) para encontrar o número x — (*ou descobrir que ele não está lá*)

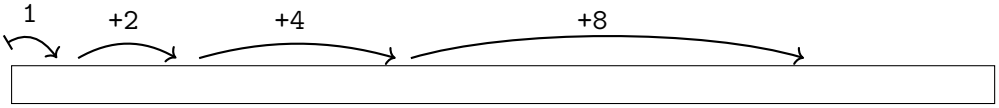
Não é legal?

3 Acelerando a busca pula-pula

Sim.

Mas, a gente ainda pode fazer melhor do que isso.

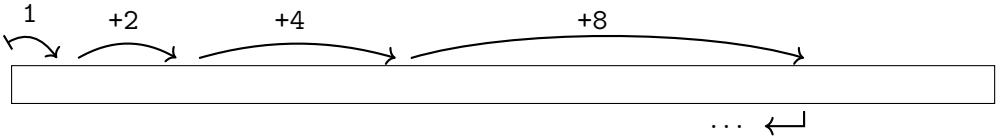
Quer dizer, a esperteza agora é a gente ir aumentando o tamanho do pulo



Dessa maneira, a gente avança bem mais rápido pela lista — (*examinando poucos elementos*)

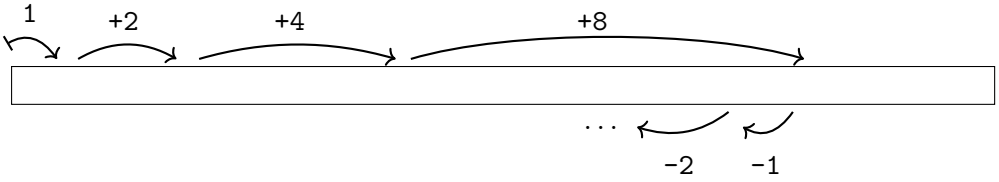
Mas o problema é que os pulos vão ficando cada vez maiores.

Daí que, na hora em que a gente passa do ponto, pode ter que voltar um montão



Mas, isso não é realmente um problema.

Porque a gente pode voltar pulando também



E se a gente passar do ponto outra vez, é só começar a pular outra vez.

Legal.

Agora é preciso implementar essa ideia.

E a melhor maneira de fazer isso é escrever uma função `Pula-pula()`.

Quer dizer, essa função vai implementar o procedimento padrão em uma faixa da lista.

E o programa principal vai utilizá-la para ficar indo e voltando.

Certo.

A função só tem dois resultados possíveis

- ou ela acha o elemento
- ou ela passa do ponto

E para indicar o que aconteceu, a gente vai utilizar 3 valores de retorno

`res, pos, pant` : resultado, posição onde a coisa parou, e posição anterior

Daí, a coisa fica assim

```
func Pula-pula ( x, L, inicio, fim, direcao )
    Se ( direção == 1 )                        // pulando para frente
        Se ( V[inicio] == x )
            Retorna 1, inicio, 0
        pant = inicio;   i = inicio+1;   pulo = 2
        Enquanto ( i ≤ fim   E   V[i] < k )
            pant = i;   i = i + pulo;   pulo = pulo * 2
        Se ( i > fim   OU   V[i] > k )
            Retorna 0, MIN(i,fim), pant        // não achou
        Senão
            Retorna 1, i, 0                    // achou
    Se ( direção == -1 )                       // pulando para trás
        (...  a mesma coisa ...  )
```

E o programa principal fica assim

```
inicio = 1;   fim = N;   direcao = 1
res = 0                                        // pulando para frente
Enquanto ( res == 0 )
    res,a,b  =  Pula-pula ( x, L, inicio, fim, direcao )
    Se ( direcao == 1 )
        fim = a;   inicio = b+1;   direção = direção * (-1)
        Se ( inicio > fim )     Quebra
    Se ( direcao == -1 )   (...  a mesma coisa ...)
```

Análise

Mas, e agora?

- *Qual é o tempo de execução desse algoritmo ?*

Bom, a função `Pula-pula()` é fácil.

Porque, a cada volta do laço, o pulo dobra de tamanho.

Logo, após $\log_2(n)$ voltas, o pulo é igual a n.

E ao dar um pulo desse tamanho, a gente cai fora da lista.

Logo, a função executa em tempo $O(\log\, n)$ — (*no pior caso*)

Certo.

Agora, a gente precisa estimar o número de voltas do laço no programa principal.

E aqui, a gente foca nas variáveis `inicio` e `fim`.

Quer dizer, quando o algoritmo começa, a distância entre elas é igual a n.

Mas daí, logo após a primeira chamada à função `Pula-pula()`, a distância é no máximo $n/2$ — (*porque?*)

E após a segunda chamada, a distância é no máximo $n/4$ — (*porque?*)

Quer dizer, a cada chamada a distância é reduzida à metade.

Logo, a coisa acaba em no máximo $\log_2(n)$ voltas.

Portanto, o algoritmo executa em tempo

$$\log_2(n) \quad \times \quad \log_2(n) \quad = \quad O(\log^2\, n)$$

Concretamente, em uma lista com 1.000.000 elementos, basta examinar $20^2 = 400$ deles (aprox.) para encontrar o número x — (*ou descobrir que ele não está lá*)

Não é legal?

4 A busca binária

Sim.

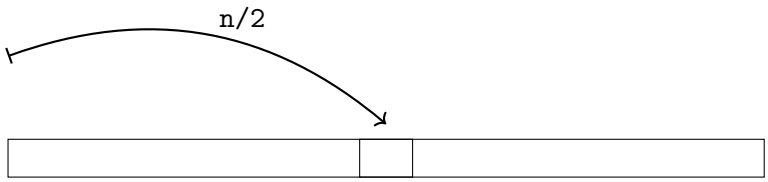
Mas, a gente ainda pode fazer melhor do que isso.

E a ideia é inverter as coisas.

Quer dizer, a busca pula-pula acelerada começa dando pulos pequenos, e daí vai aumentando o tamanho do pulo.

Agora, a gente vai começar dando pulos grandes, e daí vai diminuindo o tamanho do pulo.

A coisa começa com um pulo de tamanho $n/2$, que nos leva para o meio da lista



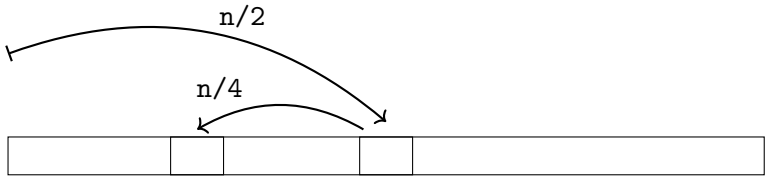
E daí, existem 3 possibilidades

1. Se esse é o elemento x que a gente procura, então a busca acaba aí
2. Se ele é maior do que x, então a gente pode descartar o lado direito
   — (*porque lá, todo mundo também é maior*)
3. Se ele é menor do que x, então a gente pode descartar o lado esquerdo
   — (*porque lá, todo mundo também é menor*)

E daí, a coisa continua assim.

Quer dizer, imagine que a gente está no caso 2 — (*onde o lado direito é descartado*)

Então agora, a gente dá um pulo para a esquerda de tamanho $n/4$



E faz a mesma coisa outra vez.

A coisa continua assim até que o elemento seja encontrado.

Ou até que a lista inteira seja descartada.

Abaixo nós temos o algoritmo da busca binária

```
inicio = 1;   fim = N;   achei = 0

Enquanto ( inicio ≤ fim )

    meio = (inicio + fim) / 2

    Se ( L[meio] == x )     achei = 1;   Quebra

    Se ( L[meio] > x )      fim = meio - 1

    Se ( L[meio] > x )      fim = meio - 1

Se ( achei == 1 )     Imprime ('Achei')

Senão                 Imprime ('Não está lá')
```

Análise

Dessa vez a análise não é difícil.

Porque a distância entre `inicio` e `fim` vai reduzindo à metade a cada volta do laço.

Logo, o laço dá no máximo $\log_2(n)$ voltas.

E o algoritmo executa em tempo

$$O(\log\, n)$$

Concretamente, em uma lista com 1.000.000 elementos, basta examinar 20 deles (aprox.) para encontrar o número x — (*ou descobrir que ele não está lá*)

Não é legal?

4.1 Implementação recursiva

A busca binária não é realmente uma esperteza.

Quer dizer, ela é a aplicação de uma ideia bastante geral: a técnica da divisão.

Como o nome já diz, a ideia é

→ *dividir o problema em duas metades, e daí resolver as duas metades*

Mas dessa vez, a situação é melhor ainda, porque a gente só precisa resolver uma metade — (*ou fazer a busca em apenas uma metade*)

Quando a gente vê as coisas assim, é mais natural escrever o algoritmo de maneira recursiva

```
func Busca-bin ( x, L, inicio, fim )

    Se ( inicio > fim )   Retorna 0

    meio = (inicio + fim) / 2

    Se ( L[meio] == x )     Retorna 1

    Se ( L[meio] > x )      resp = Busca-bin ( x, L, inicio, meio-1 )

    Se ( L[meio] < x )      resp = Busca-bin ( x, L, meio+1, fim )

    Retorna resp
```

Análise

Aqui, a gente vê que o algoritmo executa em tempo $O(\log\, n)$, porque ele faz no máximo $\log_2(n)$ chamadas recursivas — (*porque?*)

5 O algoritmo Mergesort

Agora nós vamos ver outra aplicação da técnica da divisão — (*porque essa é uma ideia realmente legal ...*)

(. . .)