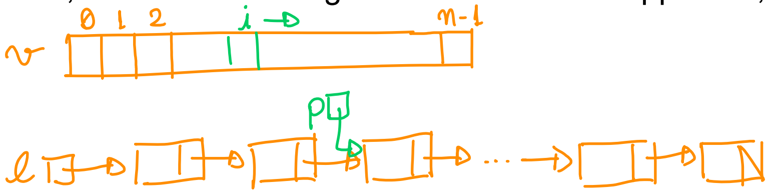
Revisão d~~e~~ as busca sequencial e binária ~~e crescimento de funções~~

"Busca binária está para algoritmos assim como a roda está para mecânica: ela é simples, elegante e imensamente importante"
- U. Manber, Introduction to Algorithms: a Creative Approach, 1989.

Busca sequencial



Para resolver este problema, temos de
- devolver a posição de um determinado elemento x em um vetor de inteiros v.

Uma ideia básica é percorrer o vetor verificando se x é o elemento de cada posição.

Algoritmo iterativo que busca um elemento x em um vetor v de tamanho n

```
int buscaSequencial1(int v[], int n, int x) {
   int i = 0;
   while (i < n && v[i] != x)
       i++;
   if (i < n) return i;
   return -1;
}
```

O(n) iterações

buscaSeq(v[ ], n, x):
  i = 0
  enquanto (i < n E v[i] ≠ x): (vetor ñ acabou; ñ achou x)
    i = i + 1
  se (i < n) devolva i
  devolva -1

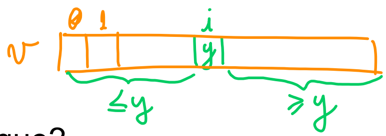
Variações:
- Uma variação do algoritmo anterior é percorrer o vetor do fim para o início.
  - Neste caso, se o elemento não for encontrado i sai do laço valendo -1,
    - o que permite eliminar o if.
- Quiz1: Embora esta variante também resolva o problema,
  - ela pode devolver valor diferente numa situação específica. Qual?

Eficiência de tempo: No pior caso o algoritmo precisa
- percorrer o vetor inteiro, realizando da ordem de n operações, i.e., O(n).

E no melhor caso? O(1)

Eficiência de espaço: O(1), pois só usa uma pequena quantidade
- de variáveis cujos tamanhos não dependem de n.

Quiz2: E se o vetor estiver ordenado, nossa busca sequencial pode ser melhorada?
- Supondo que o vetor está em ordem crescente e
  - que estamos percorrendo-o do início ao fim,
- quando encontrarmos algum valor maior que x
  - sabemos que não adianta continuar buscando. Por que?
- O seguinte algoritmo utiliza essa ideia.



se v[i] = y > x
não adianta continuar a busca

Pular daqui

Algoritmo iterativo que realiza busca sequencial de um elemento x
- em um vetor v em ordem crescente de tamanho n.

```c
int buscaSequencial2(int v[], int n, int x) {
    int i = 0;
    while (i < n && v[i] < x) i++;
    if (i < n && v[i] == x) return i;
    return -1;
}
```

Convenções e variações:

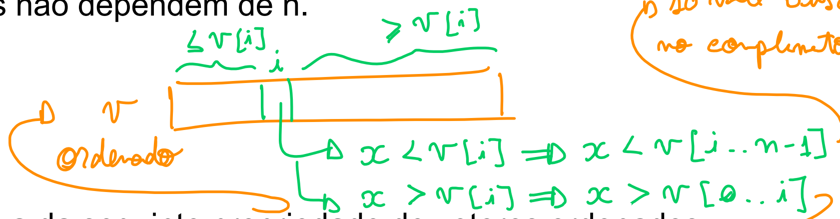
- O algoritmo anterior devolve -1 se não encontrou o elemento.
- Outra convenção válida é devolver a posição em que
  - o elemento deveria ser inserido, de modo a manter a ordenação.
- Quiz3: Como modificar o algoritmo para refletir esta convenção?

Eficiência de tempo: o número de operações no pior caso é da ordem de n,
  - ou, simplesmente, O(n),
- mas vale notar que a constante é melhor no caso médio,
  - já que, em média, após percorrer metade do vetor
    - encontramos um elemento >= x e saímos do laço.

até aqui

Eficiência de espaço: O(1), pois só usa uma pequena quantidade de variáveis,
  - cujos tamanhos não dependem de n.

→ só vale buscar x no complemento v[0..i-1]



Busca binária

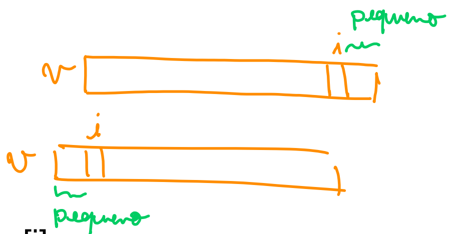
→ só vale buscar x no complemento v[i+1..n-1]

A ideia da busca binária deriva da seguinte propriedade de vetores ordenados:

- Se o valor buscado x é menor que o valor na i-ésima posição do vetor v,
  - i.e., x < v[i],
- Então x é menor que todo valor em v[i .. n - 1].
  - Portanto, x só pode ser encontrado
    - no subvetor complemento v[0 .. i - 1].
- Caso contrário, i.e., x > v[i], temos x maior que todo valor em v[0 .. i].
  - Portanto, x só pode ser encontrado
    - no subvetor complemento v[i+1..n-1].

Essa propriedade significa que,

- dependendo do índice i do valor v[i] com o qual comparamos x,
  - podemos descartar grandes pedaços do vetor.
- Por isso, devemos escolher sabiamente o índice i.

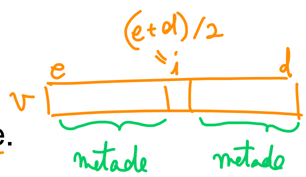


Note que, um índice i próximo dos extremos do vetor corrente

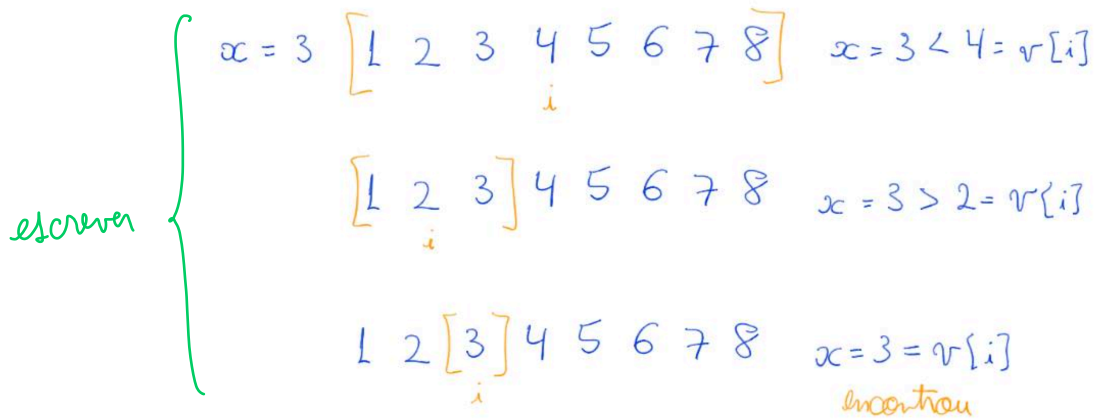
- pode resultar em descartes pequenos,
  - dependendo do resultado da comparação entre x e v[i].

Assim, o valor de i que nos garante descartes significativos,

- independente de tal resultado é i igual ao meio do vetor corrente.



Exemplo de busca binária:



Algoritmo recursivo para busca binária de um elemento x

- em um vetor v em ordem crescente de tamanho n.

```
int buscaBinariaR(int v[], int e, int d, int x) {
   int m;
   if (d < e) return -1;
   m = (e + d) / 2;
   if (v[m] == x) return m;
   if (v[m] < x) return buscaBinariaR(v, m + 1, d, x);
   return buscaBinariaR(v, e, m - 1, x);
}

int buscaBinaria2(int v[], int n, int x) {
   return buscaBinariaR(v, 0, n - 1, x);
}
```

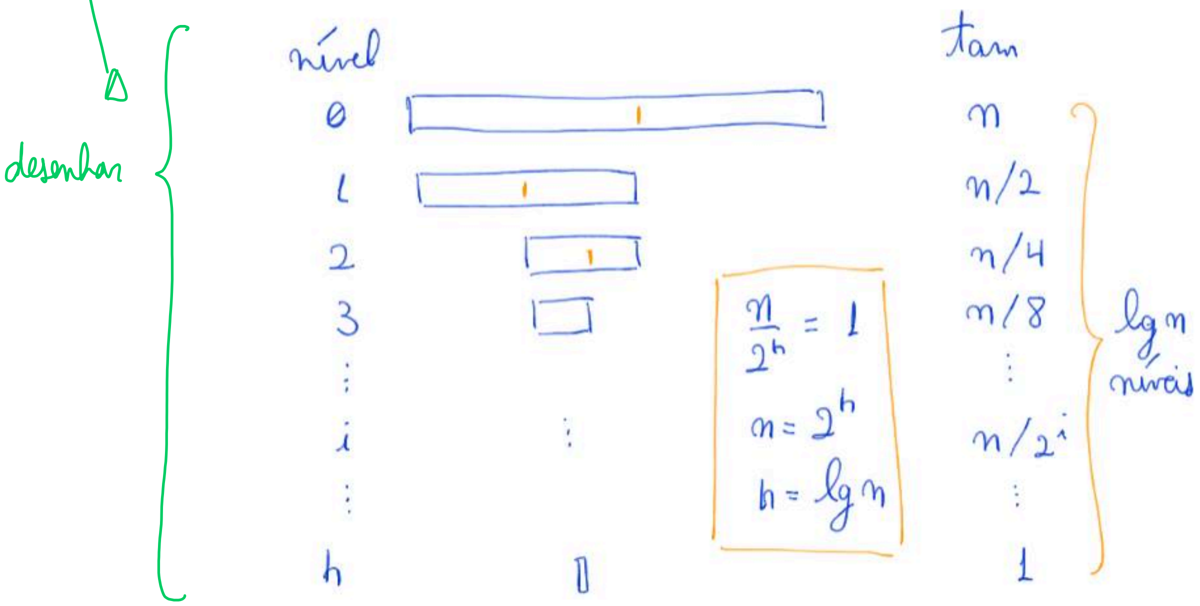
```
buscaBinR(v[], e, d, x):
  se e>d : devolva -1          — não achou
  m = (e+d)/2
  se v[m] == x : devolva m     — achou
  se v[m] < x : descer à dir.:
     devolva buscaBinR(v, m+1, d, x)
  devolva buscaBinR(v, e, m-1, x)
                               descer à esq.
```

função envelope

```
buscaBin(v, n, x):
  devolva buscaBinR(v, 0, n-1, x)
```

Eficiência de tempo: Cada chamada da função buscaBinariaR

- desencadeia no máximo uma chamada recursiva, na qual um dos extremos
  - (e ou d) é atualizado com m (+ ou - 1), sendo que m = (e + d) / 2.
- Por isso, o vetor corrente (que começa em e e termina em d)
  - diminui de pelo menos metade a cada chamada recursiva.
- Intuitivamente, podemos pensar que a cada chamada recursiva
  - o vetor que começa com tamanho n é dividido pela metade.



- Assim, depois de aproximadamente lg n chamadas recursivas,
    - seu tamanho é reduzido a 1, e as chamadas terminam.
- Como o número de operações realizadas
    - localmente em cada chamada da função é constante,
        - o algoritmo leva tempo da ordem de lg n, ou, O(lg n).

topo
⋮
3ª cha
2ª cha
1ª cha

Eficiência de espaço:
- O(log n), devido ao tamanho da cadeia de chamadas recursivas. (altura da pilha de recursão)

Algoritmo iterativo para busca binária de um elemento x
- em um vetor v em ordem crescente de tamanho n.

```
int buscaBinaria(int v[], int n, int x) {
   int e, m, d;
   e = 0;
   d = n - 1;
   while (e <= d) {
       m = (e + d) / 2;
       if (v[m] == x)
           return m;
       if (v[m] < x) e = m + 1;
       else d = m - 1;
   }
   return -1;
}
```

```
buscaBin (v[], n, x):
  e = 0; d = n-1;
  enquanto (e ≤ d):
    m = (e+d)/2
    se (v[m] = x): devolva m — achou
    se v[m] < x: e = m+1 — desce à dir.
    senão d = m-1 — desce à esq.
  devolva -1 — não achou
```

Eficiência de tempo: O(log n), sendo que a demonstração
- é igual aquela feita para o algoritmo recursivo,
    - substituindo chamada recursiva por iteração na argumentação.

Eficiência de espaço: O(1), pois só usa uma pequena quantidade
- de variáveis auxiliares, cujos tamanhos não dependem de n.

Tabela de símbolos

Também é chamada de dicionário.
- Corresponde a um conjunto de itens,
    - em que cada item possui uma chave e um valor.
- Suporta diversas operações sobre os itens,
    - sendo busca a principal delas.
- Trata-se de um Tipo Abstrato de Dado, pois
    - o foco está no propósito da estrutura, e não em sua implementação.

Estamos interessados nas seguintes operações:
- busca - dada uma chave k, devolva um apontador
    - para um objeto com esta chave. Se não existir devolva “none”.

- min (max) - devolva um apontador
  - para um objeto com a menor (maior) chave.
- predecessor (sucessor) - dada uma chave k, devolva um apontador
  - para o objeto com a maior (menor) chave menor (maior) que k.
    - Se não existir devolva "none".
- percurso ordenado - devolva todos os objetos
  - seguindo a ordem de suas chaves.
- seleção - dado um inteiro i, entre 1 e n, devolva um apontador
  - para o objeto com a i-ésima menor chave.
- rank - dada uma chave k, devolva o número de objetos
  - com chave menor ou igual a k.

Curiosidade: Note que as posições na seleção vão de 1 a n, não de 0 a n - 1.
- Isso faz as operações de rank e seleção serem inversas entre si.

Vamos começar a pensar na implementação de uma tabela de símbolos
- e nas estruturas de dados que podemos usar para tanto.

Implementação em vetor ordenado

Considere um vetor ordenado v de tamanho n.
- Como podemos implementar as operações anteriores?

Exemplificar operações com o seguinte vetor
- 3 6 10 11 17 23 30 36

percurso ordenado: 3 6 10 11 17 23 30 36

busca(8): usando busca binária temos

limite esq. → meio ← limite dir.

| 3 6 10 11 17 23 30 36 |

1º passo: 8 ≤ 11 ⇒ buscar no subvetor à esquerda

limite esq. → meio ← limite dir.

| 3 6 10 11 | 17 23 30 36 |

2º passo: 8 > 6 ⇒ buscar no subvetor à direita

limite esq. → meio ← limite dir.

| 3 6 | 10 11 | 17 23 30 36 |

3º passo: 8 ≤ 10 ⇒ buscar no subvetor à esquerda

lim. esq. → ← lim. dir.

| 3 6 | 10 | 11 17 23 30 36 |

Caso parada: limite dir. = limite esq. + 1 min: [3 6 10 11 17 23 30 36]

predecessor(10): usando busca binária temos

[3 6 10 11 17 23 30 36]

localiza o elemento

[3 6 10 11 17 23 30 36]

devolve o anterior, se não cair fora do vetor

1	2	3	4	5	6	7	8
3	6	10	11	17	23	30	36

seleção(7): [3 6 10 11 17 23 30 36]

rank(12): usando busca binária

[3 6 10 11 17 23 30 36]

encontra posição em que a chave deveria estar

[3 6 10 11 | 17 23 30 36]

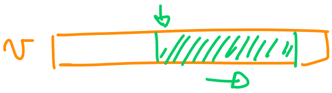
devolve número de elementos à esq. da posição, no caso 4

Eficiência das operações:

- busca - O(log n), deriva da busca binária.
- min (max) - O(1).
- predecessor (sucessor) - O(log n), deriva da busca binária.
- percurso ordenado - O(n), mínimo possível já que é o tamanho da saída.
- seleção - O(1).
- rank - O(log n), deriva da busca binária.

Quiz4: Qual é então o ponto fraco dos vetores ordenados?

- Não são eficientes quando o conjunto de itens é dinâmico.
  - inserção - O(n). Por que?
  - remoção - O(n). Por que?



Árvores binárias de busca balanceadas combinam características
  - dos vetores ordenados e das listas ligadas
- para implementar tabelas de símbolos
  - que trabalham com conjuntos dinâmicos.