AED1 - Aula 06
Crescimento de funções, busca sequencial e binária em vetores

"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.

Crescimento de funções

n	10^3	10^6	10^9
log_2 n	10	20	30
n^(1/2)	32	10^3	3*10^4
n log_2 n	10^4	2*10^7	3*10^10
n^2	10^6	10^12	10^18
n^3	10^9	10^18	10^27
2^n	10^300	10^300000	10^(3*10^8)

Interpretação temporal considerando um computador de 1GHz

n	10^3	10^6	10^9
log_2 n	<< 1s	<< 1s	<< 1s
n^(1/2)	<< 1s	<< 1s	<< 1s
n	<< 1s	<< 1s	1s
n log_2 n	<< 1s	< 1s	30s
n^2	<< 1s	16 min	31 anos
n^3	1s	31 anos	31709791 milênios
2^n	esquece...		

Busca sequêncial

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;
}
```

Variações:

- uma variação do algoritmo anterior seria percorrer o vetor do fim para o início.
  - neste caso, se o elemento não for encontrado i sai do laço valendo -1.
    - isso permite eliminar o if.

Corretude e invariante:

- o invariante principal do algoritmo é que, no início de toda iteração do laço, o vetor v[0..i-1] não contém x.
  - no início o invariante vale trivialmente, pois i=0 e v[0..i-1] é vazio.
  - o invariante se mantém de uma iteração para outra pois i só é incrementado se v[i] != x.
- se o algoritmo sair do laço por violar a primeira condição (i < n), temos que
  - o invariante garante que x não está no vetor
- caso contrário a violação da segunda condição (v[i] != x)
  - garante que o algoritmo devolve a solução correta.

Eficiência:

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

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

```
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 estar, de modo a manter a ordenação.
        - isso pode ser útil, por exemplo, para inserção.
        - como modificar o algoritmo para refletir esta convenção?

Corretude e invariante:

- o invariante principal do algoritmo é que, no início de toda iteração do laço, vale que v[i-1] < x.
    - note que, como o vetor é ordenado, isso implica que todo elemento em v[0..i-1] é menor que x.
    - o invariante vale trivialmente no início, pois i começa valendo 0.
    - supondo que ele vale no início de uma iteração qualquer, como o laço só incrementa i se v[i] < x, ele continua valendo no início da próxima iteração.
- quando o algoritmo sai do laço, temos que i indica onde x deveria estar,
    - pois todo elemento em v[0..i-1] é menor que x e o algoritmo só sai do laço caso
        - o vetor tenha terminado, i.e., i = n
        - ou v[i] >= n.
- se i é um índice válido do vetor e x = v[i], o algoritmo devolve i indicando sucesso na busca.
- senão devolve -1, indicando que x não está no vetor.

Eficiência:

- o número de operações no pior caso é da ordem de n, ou simplesmente O(n).

Busca binária

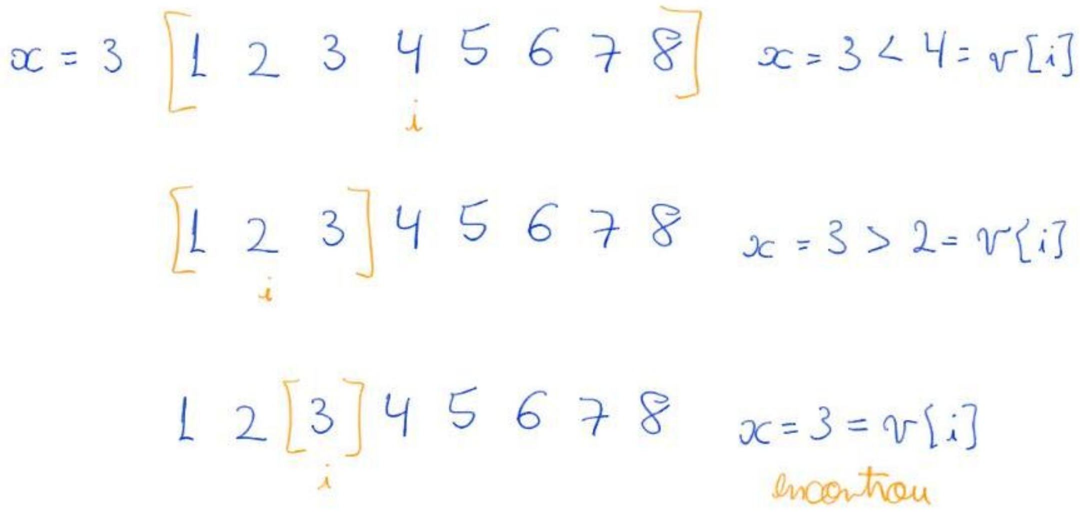
a ideia da busca binária deriva da seguinte propriedade de vetores ordenados:

- 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]
        - e portanto x só pode ser encontrado no complemento v[0..i-1].
- caso contrário,
    - i.e., x > v[i],
    - então x é maior que todo valor em v[0..i]
        - e portanto x só pode ser encontrado no 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 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 ordenado v de tamanho n.

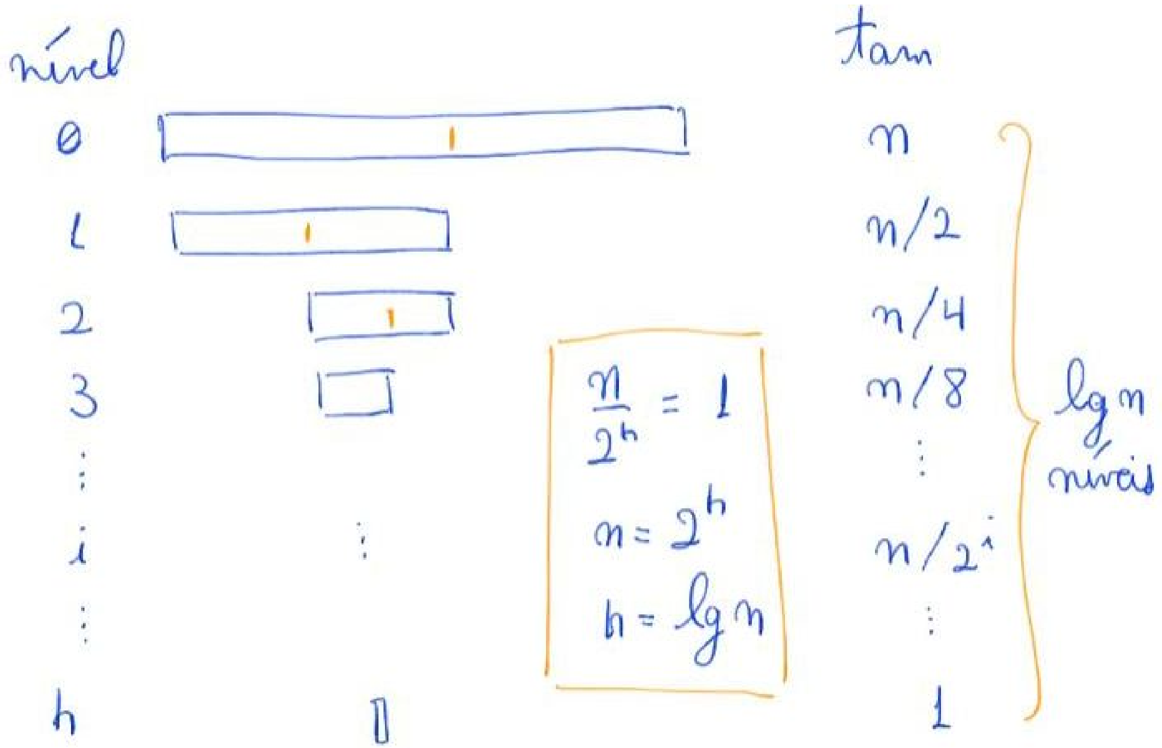
```
int buscaBinariaR(int v[], int e, int d, int x)
{
   int m;
   if (d < e)
       return -1;
   m = (e + d) / 2;
   if (v[m] == x)
       return m;
   if (v[m] < x)
       return buscaBinariaR(v, m + 1, d, x);
   return buscaBinariaR(v, e, m - 1, x);
}

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

Corretude:

- prova por indução.

Eficiência:

- 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).



- para de fato calcular o número total de operações, usamos a recorrência

  T(n) = T(n/2) + 1

  T(0) = 1
- seguindo a recorrência

  T(n/2) = T(n/4) + 1

  T(n/4) = T(n/8) + 1

  T(n/8) = T(n/16) + 1

- substituindo e simplificando

  $T(n) = T(n/2^1) + 1$
  $= T(n/2^2) + 2$
  $= T(n/2^3) + 3$
  $= T(n/2^4) + 4$
- esta relação sugere a fórmula geral

  $T(n) = T(n/2^k) + k$
- sabemos que a recorrência acaba quando o vetor corrente tiver tamanho 0
  - como as sucessivas divisões por 2 no tamanho do vetor são arredondadas para baixo, isso ocorre para k que satisfaça

    $n/2^k < 1 <= n/2^{(k-1)}$
- portanto

  $n/2^k < 1 <= n/2^{(k-1)} \rightarrow n < 2^k <= 2n \rightarrow \lg n < k <= \lg 2n = 1 + \lg n$
- substituindo k por 1 + lg n na recorrência

  $T(n) = T(n/2^k) + k = T(0) + \lg n + 1 = \lg n + 2$
- assim, o número de operações no pior caso é da ordem de lg n,
  - ou simplesmente O(lg n).

Algoritmo iterativo para busca binária de um elemento x em um vetor ordenado v 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;
}
```

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 estar, de modo a manter a ordenação.
    - isso pode ser útil, por exemplo, para inserção.
    - como modificar o algoritmo para refletir esta convenção?

Corretude e invariante:

- o invariante principal é que no início de cada iteração
  $v[e-1] < x < v[d+1]$
- adotamos a convenção de que $v[-1] = -$ infinito e $v[n] = +$ infinito.
  - portanto, o invariante vale no início da primeira iteração.
- supondo que o invariante vale no início de uma iteração qualquer,
  - como no laço só atualizamos o extremo (e ou d) que não contém x,
    - de acordo com o resultado da comparação $v[m] < x$,
  - o invariante continua valendo no início da iteração seguinte,
    - ou seja, sempre descartamos o subvetor correto.
- se em alguma iteração a comparação v[m] == x for verdadeira, devolve a posição de x,
- senão sai do laço quando $d = e + 1$
  - pelo invariante, neste caso, $v[e-1] = v[d] < x < v[d+1]$
    - portanto x não está no vetor e o algoritmo devolve -1.

Eficiência:

- igual à demonstração feita para o algoritmo recursivo,
  - substituindo chamada recursiva por iteração na argumentação.