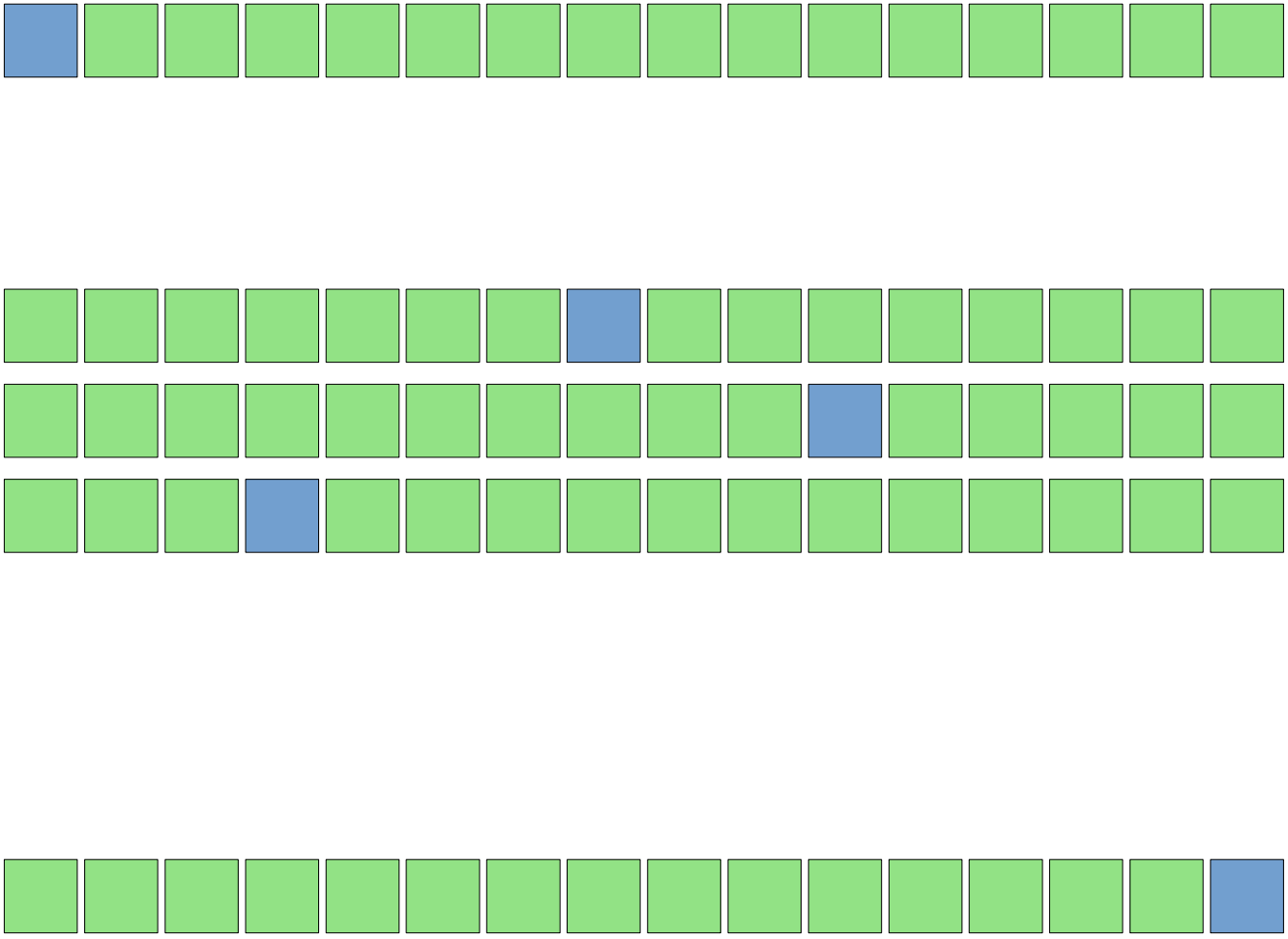
Aula 15:
- Métodos simples de busca
- Métodos simples de ordenação

Prof. João Henrique Kleinschmidt
Material elaborado pelo prof. Jesús P. Mena-Chalco
3Q-2018

Busca de um elemento em um vetor



```c
int buscaChave(int chave , int A[], int n) {
    int i;

    for(i=0; i<n; i++) {
        if (chave == A[i])
            return i;
    }
    return -1;
}
```

(*) Fonte: P. Feofiloff. Algoritmos em Linguagem C. 1ª Edição, Editora Campos, 2008.

Busca Linear

```
int buscaChave2(int chave , int A[], int n) {
    int i;
    i = n-1;

    while(i>=0 && A[i]!=chave) {
        i-=1;
    }
    return i;
}
```

Versão Elegante!

Busca Linear Recursiva

Crie uma função recursiva para identificar um elemento em um vetor.

Assinatura / Protótipo:

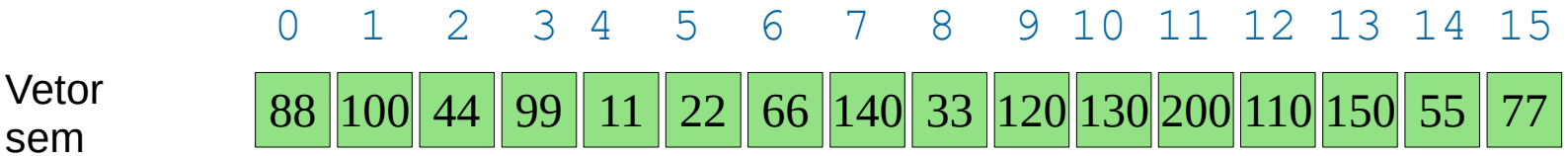
```
int buscaChaveRec (int chave, int A[], int n)
```

Busca Linear Recursiva

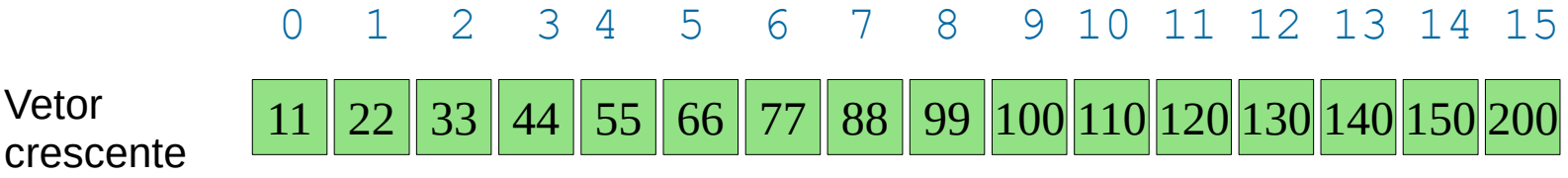
```
int buscaChaveRec (int chave, int A[], int n) {
    if (n==0)
        return -1;
    if (chave==A[n-1])
        return n-1;

    return buscaChaveRec(chave, A, n-1);
}


int main(int argc, char *argv[])
{
    int i;
```

Busca Binária

É um algoritmo de busca em vetores que segue o paradigma de divisão e conquista. Ela parte do pressuposto de que o vetor está ordenado e realiza sucessivas divisões do espaço de busca comparando o elemento buscado (chave) com o elemento no meio do vetor. Se o elemento do meio do vetor for a chave, a busca termina com sucesso. Caso contrário, se o elemento do meio vier antes do elemento buscado, então a busca continua na metade posterior do vetor. E finalmente, se o elemento do meio vier depois da chave, a busca continua na metade anterior do vetor.



Melhor caso: 1
Pior caso: n



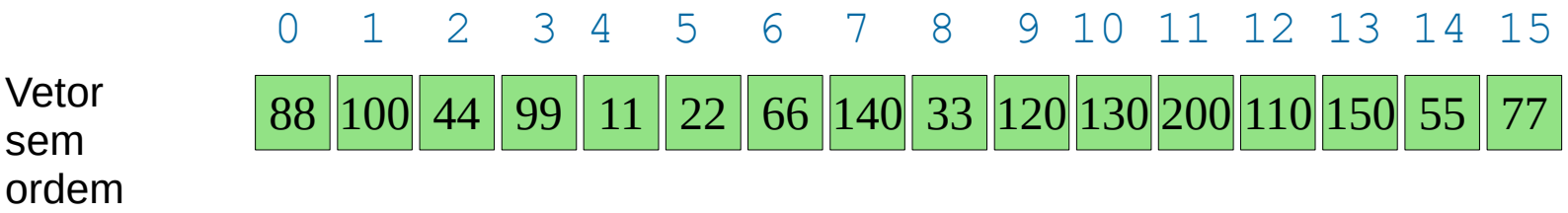
Melhor caso: 1
Pior caso: log(n)

Busca Binária Recursiva

```
int buscaBinariaRec (int chave, int A[], int inf, int sup) {
    int meio = (inf+sup)/2;

    if (inf>sup)
        return -1;

    if (chave==A[meio])
        return meio;

    if (chave<A[meio])
        return buscaBinariaRec(chave, A, inf, meio-1);
    else
        return buscaBinariaRec(chave, A, meio+1, sup);
}
```

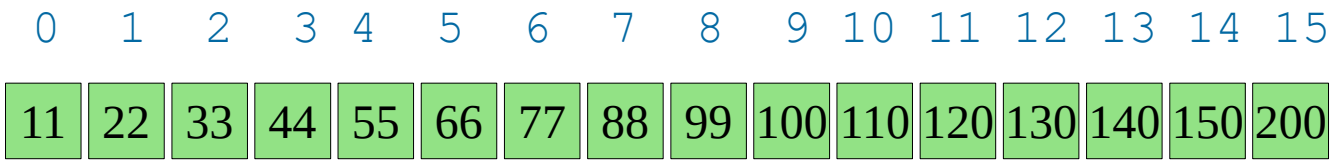
Melhor caso: 1
Pior caso: log(n)

```
buscaBinariaRec(9000000, vetor, 0, n-1) );
```

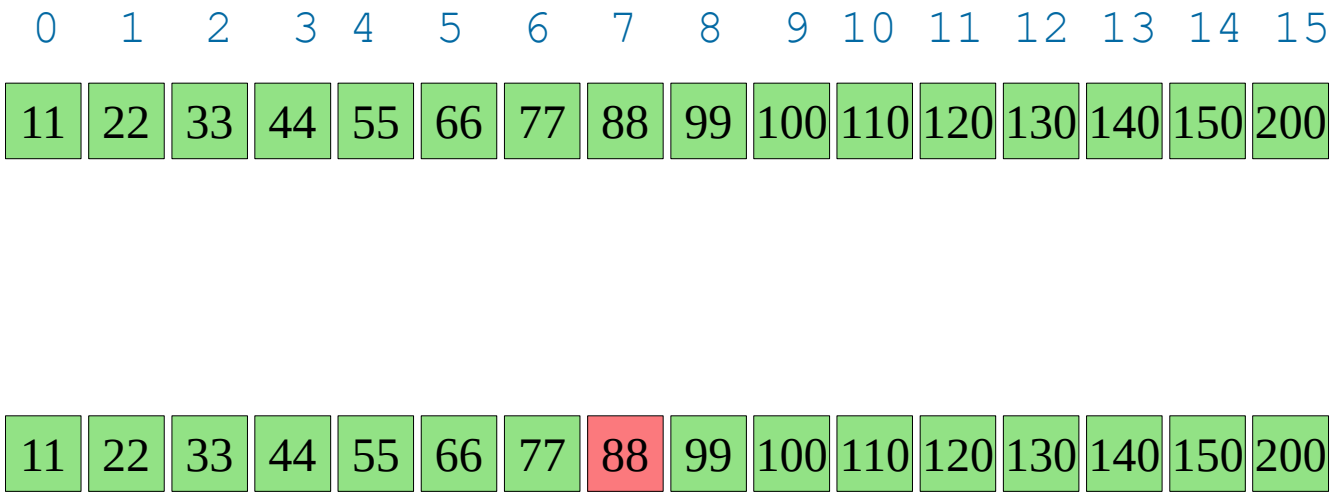


Melhor caso: 1
Pior caso: n

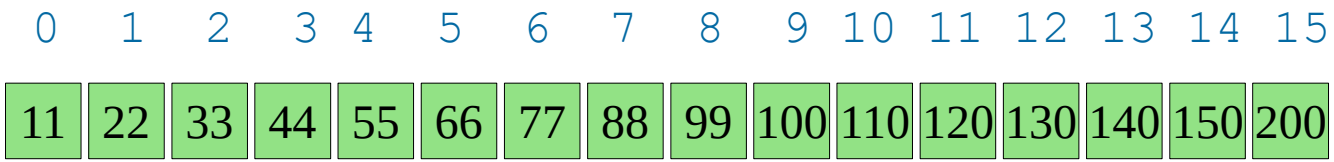
Chave = 101

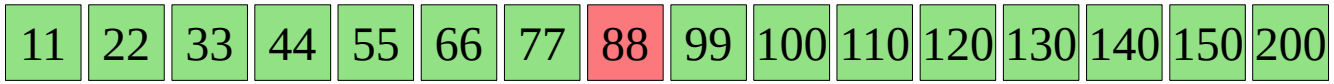

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
11 22 33 44 55 66 77 88 99 100 110 120 130 140 150 200

Chave = 101

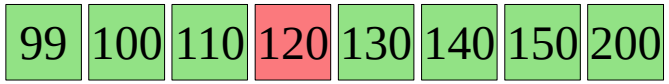

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
11 22 33 44 55 66 77 88 99 100 110 120 130 140 150 200
11 22 33 44 55 66 77 88 99 100 110 120 130 140 150 200
Inf = 0
Sup = 15

Chave = 101



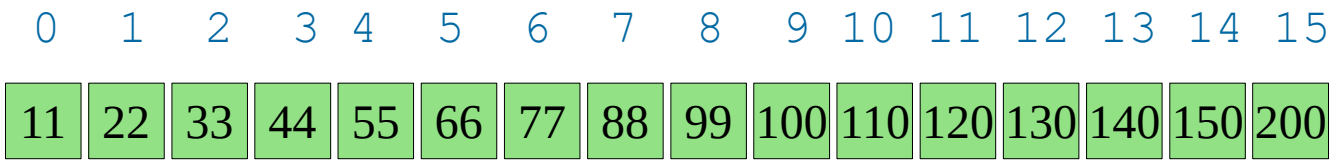


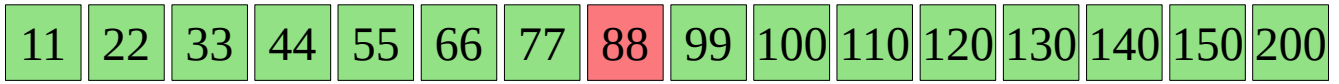
Inf = 0
Sup = 15



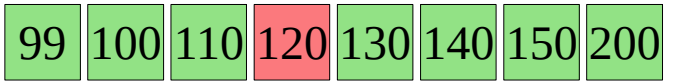
Inf = 8
Sup = 15

Chave = 101





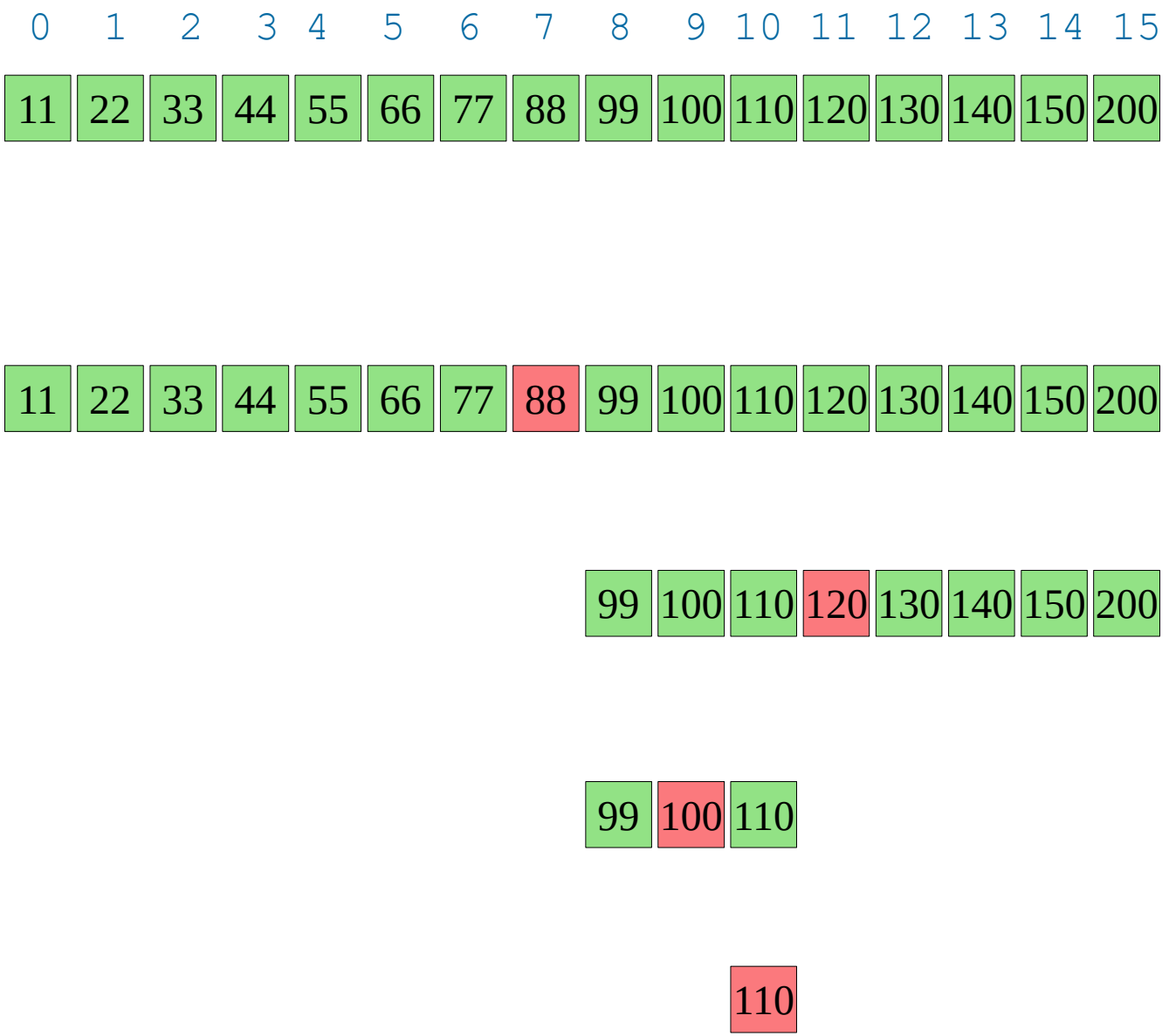
Inf = 0
Sup = 15



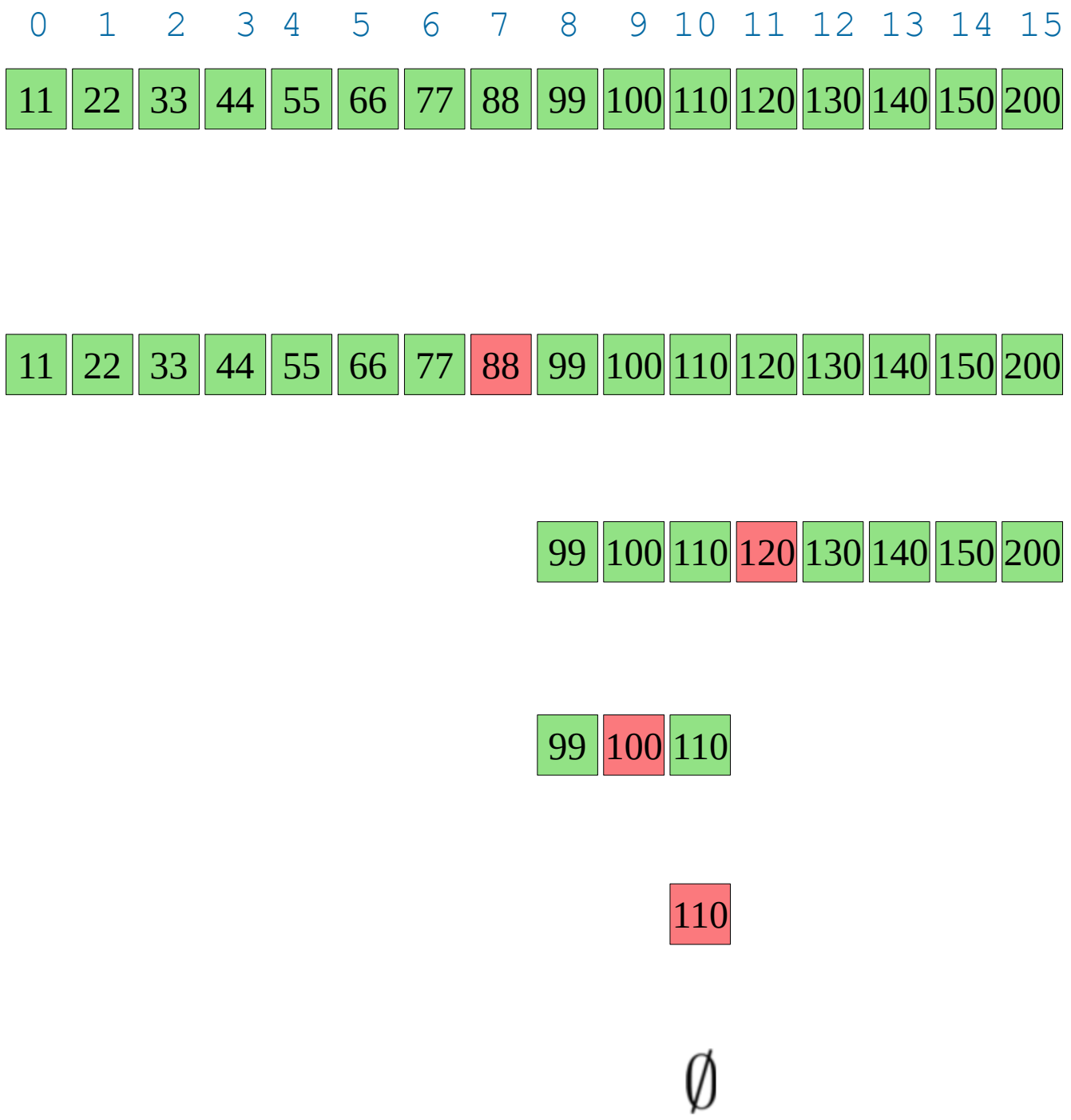
Inf = 8
Sup = 15

Inf = 8
Sup = 10

Chave = 101


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
11 22 33 44 55 66 77 88 99 100 110 120 130 140 150 200
11 22 33 44 55 66 77 88 99 100 110 120 130 140 150 200
Inf = 0
Sup = 15
99 100 110 120 130 140 150 200
Inf = 8
Sup = 15
99 100 110
Inf = 8
Sup = 10
110
Inf = 10
Sup = 10

Chave = 101


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
11 22 33 44 55 66 77 88 99 100 110 120 130 140 150 200
11 22 33 44 55 66 77 88 99 100 110 120 130 140 150 200
Inf = 0
Sup = 15
99 100 110 120 130 140 150 200
Inf = 8
Sup = 15
99 100 110
Inf = 8
Sup = 10
110
Inf = 10
Sup = 10
∅
Inf = 10
Sup = 9

Para pensar...

A) Se a chave não estiver no vetor, então devolver o índice do elemento mais próximo à chave.

B) Faça uma função iterativa para a busca binária

Ordenação

- Ordenar corresponde ao **processo de re-arranjar um conjunto de objetos** em ordem ascendente ou descendente.

- Por que ordenar?
- O objetivo principal da ordenação **é facilitar a recuperação posterior** de itens do conjunto ordenado.
- Geralmente considerado no primeiro passo para resolver um problema prático.

- As ordens mais utilizadas são a numérica e lexicográfica.

Por que ordenar?

- **Busca de elementos:**
- O usuário geralmente quer os dados ordenados.

- **Deduplicação de elementos:**
- Ordenar é muito útil para eliminar duplicações.

- Projeção de objetos em um espaço 3D.

Algoritmos para ordenar elementos

- Baseado em comparações:
- Bogo sort
- Selection sort
- Insertion sort
- Bubble sort
- Mergesort
- Quicksort
- Heapsort

- Ordenação em tempo linear:
- Radix sort

- Ordenação de primeiros elementos (seleção parcial):
- Partial Quicksort

O problema de ordenar na forma crescente

- Um vetor v[0..n-1] é crescente se v[0] ≤ v[1] ≤ ... ≤ [n-1]

- O problema de ordenação de um vetor consiste em **rearranjar (ou seja, permutar) os elementos de um vetor v[0..n-1] de tal modo que ele se torne crescente.**

- **Vetores ordenados na forma crescente:**
- {1, 1, 1, 1, 1, 1, 1, 1}
- {0, 1, 1, 1, 2, 3, 4, 4, 4, 4, 4, 4, 100}

Verificar se um vetor v[0..n-1] é crescente

```c
int crescente(int v[], int n) {
    int i;

    for (i=0; i<n-1; i=i+1)
        if (v[i]>v[i+1])
            return 0;

    return 1;
}
```

```c
int crescente2(int v[], int n) {
    int i=0;

    while(i<n-1 && v[i]<=v[i+1]) {
        i = i+1;
    }

    if (i==n-1)
        return 1;
    else
        return 0;
}
```

Número de comparações T(n):
- No melhor caso: T(n) = 1
- No pior caso: T(n) = n-1

T(n) = O(n)

Crie uma versão recursiva da função crescente

Atividade em aula

Vetor crescente (recursiva)

```c
int crescenteRec(int v[], int n) {
    if (n==1)
        return 1;

    if (v[n-2]<=v[n-1])
        return crescenteRec(v, n-1);
    else
        return 0;
}
```

```c
int main()
{
    int v[] = {0, 1, 1, 1, 2, 3, 4, 4, 4, 4, 4, 4, 100};
    int n=sizeof(v)/sizeof(v[0]);

    printf("%d\n", crescenteRec(v, n) );
}
```

Número de comparações T(n):
- No melhor caso: T(n) = 1
- No pior caso: T(n) = n-1

T(n) = O(n)

Embaralhando um vetor

```c
#include <time.h>
#include <stdio.h>
#include <stdlib.h>

void embaralhar(int v[], int n) {
    int i, r, temp;
    srand ( time(NULL) );

    for (i=0; i<n; i=i+1) {
        r = rand()%n;

        temp = v[i];
        v[i] = v[r];
        v[r] = temp;
    }
}
```

Número de trocas de elementos T(n):
- No melhor caso: T(n) = n
- No pior caso: T(n) = n

T(n) = O(n)

(1)
Bogo sort
Miracle sort
Monkey sort
Stupid sort

```c
void imprimir(int v[], int n) {
    int i;
    for (i=0; i<n; i=i+1) {
        printf("%d ", v[i] );
    }
    printf("\n");
}
```

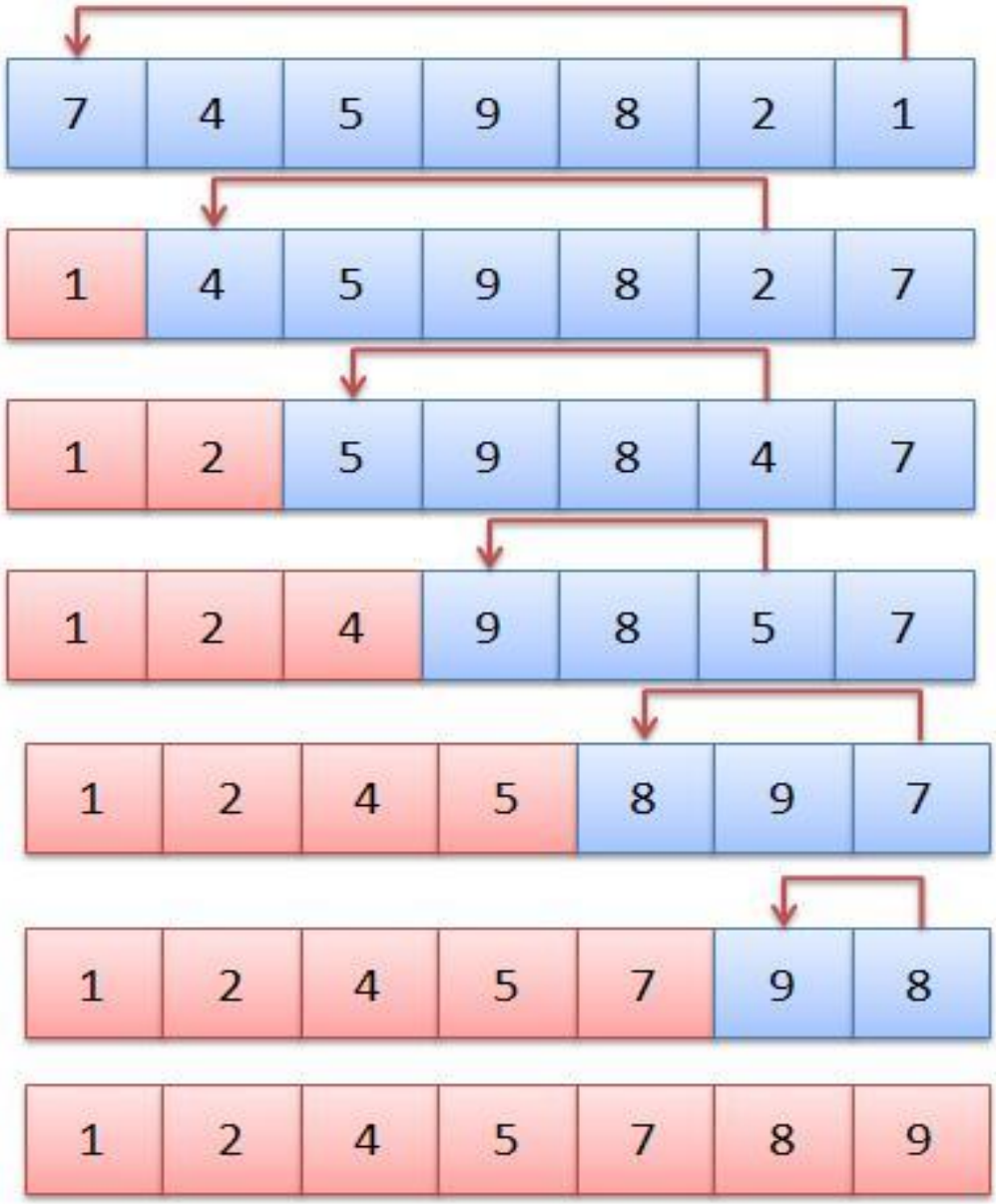
```c
void bogoSort(int v[], int n) {
    while (crescenteRec(v,n)==0) {
        embaralhar(v,n);
    }
}
```

```c
int main()
{
    int v[] = {100,4,999,555,222,3,0,-1};
    int n=sizeof(v)/sizeof(v[0]);

    imprimir(v, n);
    bogoSort(v, n);
    imprimir(v, n);
}
```

Número de comparações T(n):
- No melhor caso: T(n) = n-1
- No pior caso: T(n) = ?

(2)
Selection Sort:
Algoritmo de ordenação por seleção

Selection Sort



Animação: https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html

Selection Sort

```
void SelectionSort (int v[], int n) {
    int i, j, iMin, aux;

    for (i=0; i<n-1; i=i+1) {
        iMin = i;

        for (j=i+1; j<n; j=j+1) {
            if (v[iMin]>v[j])
                iMin = j;
        }

        if (iMin!=i) {
            aux     = v[iMin];
            v[iMin] = v[i];
            v[i]    = aux;
        }
    }
}
```

Complexidade computacional: No pior caso? $O(n^2)$