Busca e Ordenação

Ordenar e Buscar

Muitas tarefas necessitam da operação de ordenação e busca:

- Ordenar os itens recomendáveis
- Buscar uma página da Web

Isso vai além de exemplos visuais.

UFABC

Busca

Se temos um vetor de **n** elementos sem uma ordem específica, a busca por um elemento necessita de **1** a **n** operações:

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

Busca

Para uma busca eficiente precisamos estruturar nossos dados de uma forma que os conteúdos estejam ordenados.

Vamos aprender a ordenar e logo mais retornamos a busca.

UFABC

Ordenação ingênua

Uma das formas mais ingênuas de ordenar uma lista é utilizando o algoritmo BubbleSort:

```
Para i de 0 até n,
  Para j de 0 até n-1,
    Se v[j] > v[j+1], troque v[j] por v[j+1]
```

UFABC

Ordenação ingênua

Quantas instruções devemos fazer?

Para i de 0 até n,
  Para j de 0 até n-1,
    Se v[j] > v[j+1], troque v[j] por v[j+1]

Repetimos n vezes n-1 comparações:

$n \cdot (n - 1) = n^2 - n$

UFABC

Ordenação ingênua

A ideia é que a cada repetição, o maior elemento seja empurrado para o final da lista!

A cada repetição, precisamos ir até o final?

Ordenação ingênua

Para i de 0 até n,

  Para j de 0 até n-1-i,

    Se v[j] > v[j+1], troque v[j] por v[j+1]

Ordenação ingênua

Agora fazemos "apenas" $(n^2 - n)/2$ operações.

Para i de 0 até n,
  Para j de 0 até n-1-i,
    Se v[j] > v[j+1], troque v[j] por v[j+1]

UFABC

Ordenação ingênua

Se em uma passagem não efetuamos nenhuma troca, o que isso significa?

UFABC

Ordenação ingênua

Para i de 0 até n,

  Para j de 0 até n-1-i,

    Se v[j] > v[j+1], troque v[j] por v[j+1]

  Se não houve troca, pare!

Ordenação ingênua

Se dermos sorte de a lista estar ordenada, bastam **n** comparações (mas no pior caso ainda temos $(n^2 - n)/2$).

Para i de 0 até n,

  Para j de 0 até n-1-i,

    Se v[j] > v[j+1], troque v[j] por v[j+1]

  Se não houve troca, pare!

UFABC

Insertion Sort

Similar ao ingênuo, a ideia principal é inserir cada elemento em sua posição correta:

```
Para i de 1 até n,
  Para j de i até 0 e enquanto v[j-1] > v[j],
    Troca v[j-1] com v[j]
```

UFABC

Insertion Sort

No pior caso, repetimos (n-1) a passagem de i até 0, o que é similar ao Bubble Sort: $(n^2 - n)/2$ operações de troca.

Para i de 1 até n,

  Para j de i até 0 e enquanto v[j-1] > v[j],

    Troca v[j-1] com v[j]

UFABC

Insertion Sort

No melhor caso, repetimos (n-1) operações, pois a condição v[j-1] > v[j] sempre irá falhar na primeira tentativa.

Para i de 1 até n,

  Para j de i até 0 e enquanto v[j-1] > v[j],

    Troca v[j-1] com v[j]

Insertion Sort

É mais rápido para listas quase ordenadas do que o Bubble Sort (faça uma simulação!).

É interessante para listas ligadas e fluxo contínuo de dados.

Quicksort

Algoritmo de ordenação baseado na estratégia recursiva **dividir e conquistar**.

A ideia geral é, dado um vetor a ser ordenado, escolhemos um elemento para servir de **pivô** e dividimos o vetor em dois vetores menores, um com elementos menor que o pivô e outro com elementos maiores que ele.

Quicksort

O processo é repetido recursivamente até que a partição contenha apenas um elemento, e a solução se torna trivial.

UFABC

Quicksort

6	1	4	5	9	2	3

UFABC

Quicksort

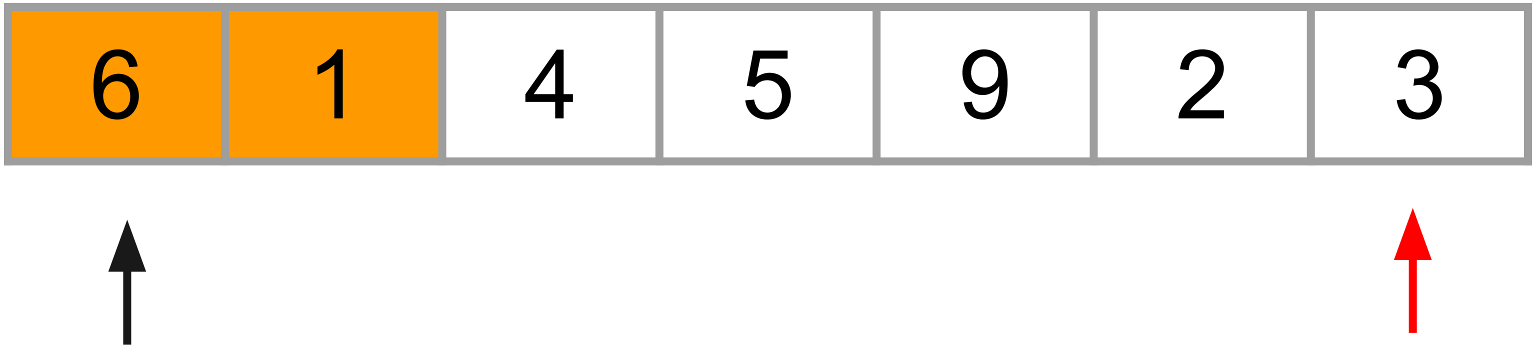
6	1	4	5	9	2	3

PIVÔ

Quicksort

6	1	4	5	9	2	3

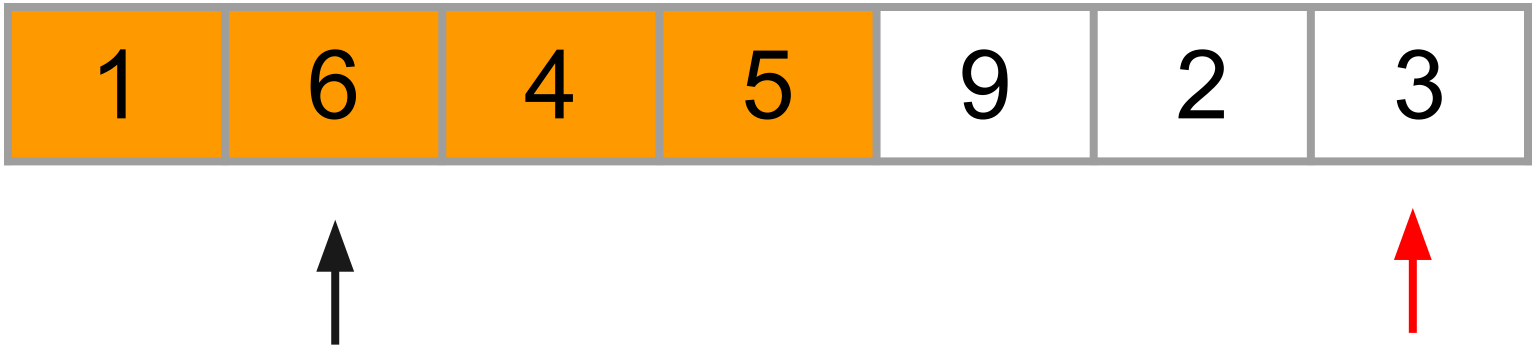
Quicksort



Quicksort

1	6	4	5	9	2	3

Quicksort

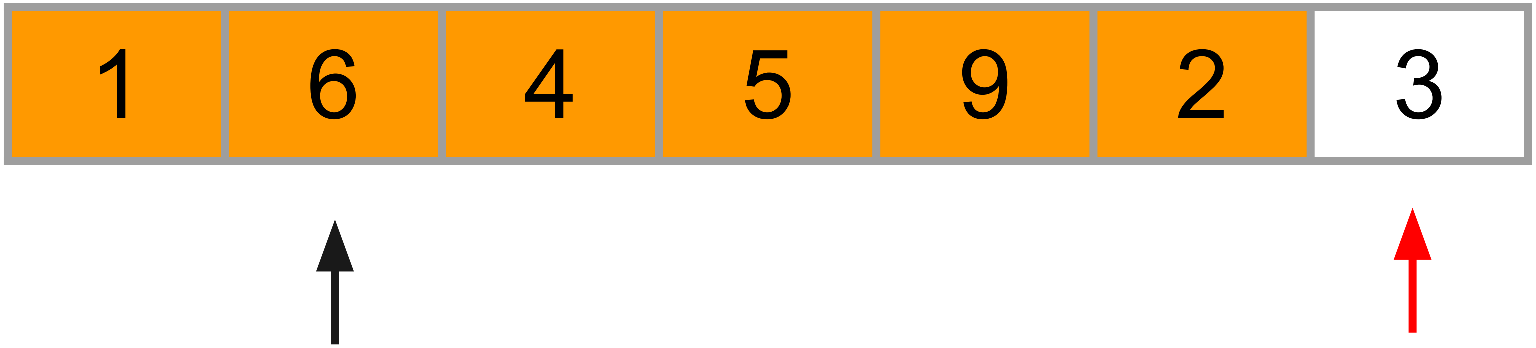


Quicksort

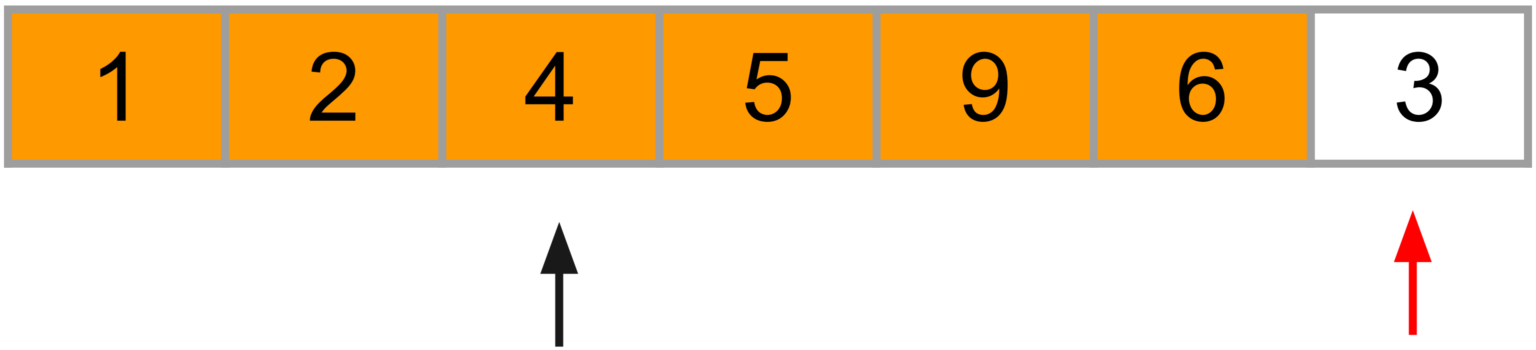
1	6	4	5	9	2	3

UFABC

Quicksort



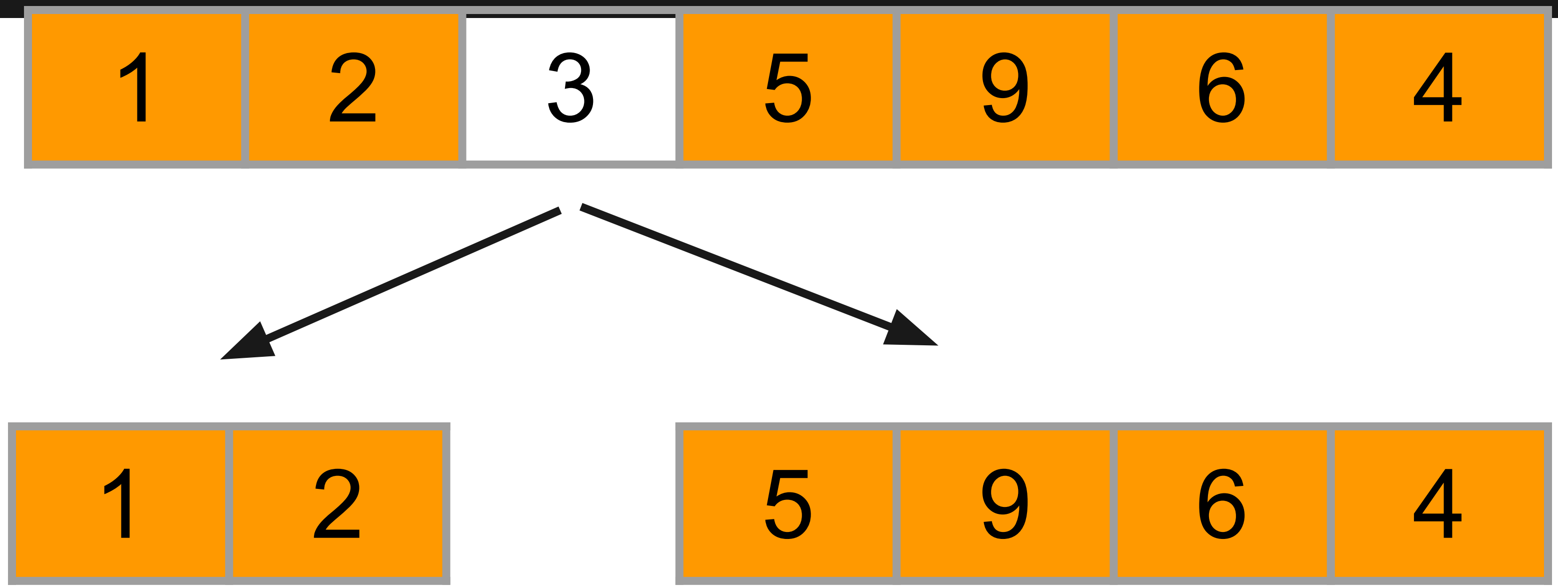
Quicksort



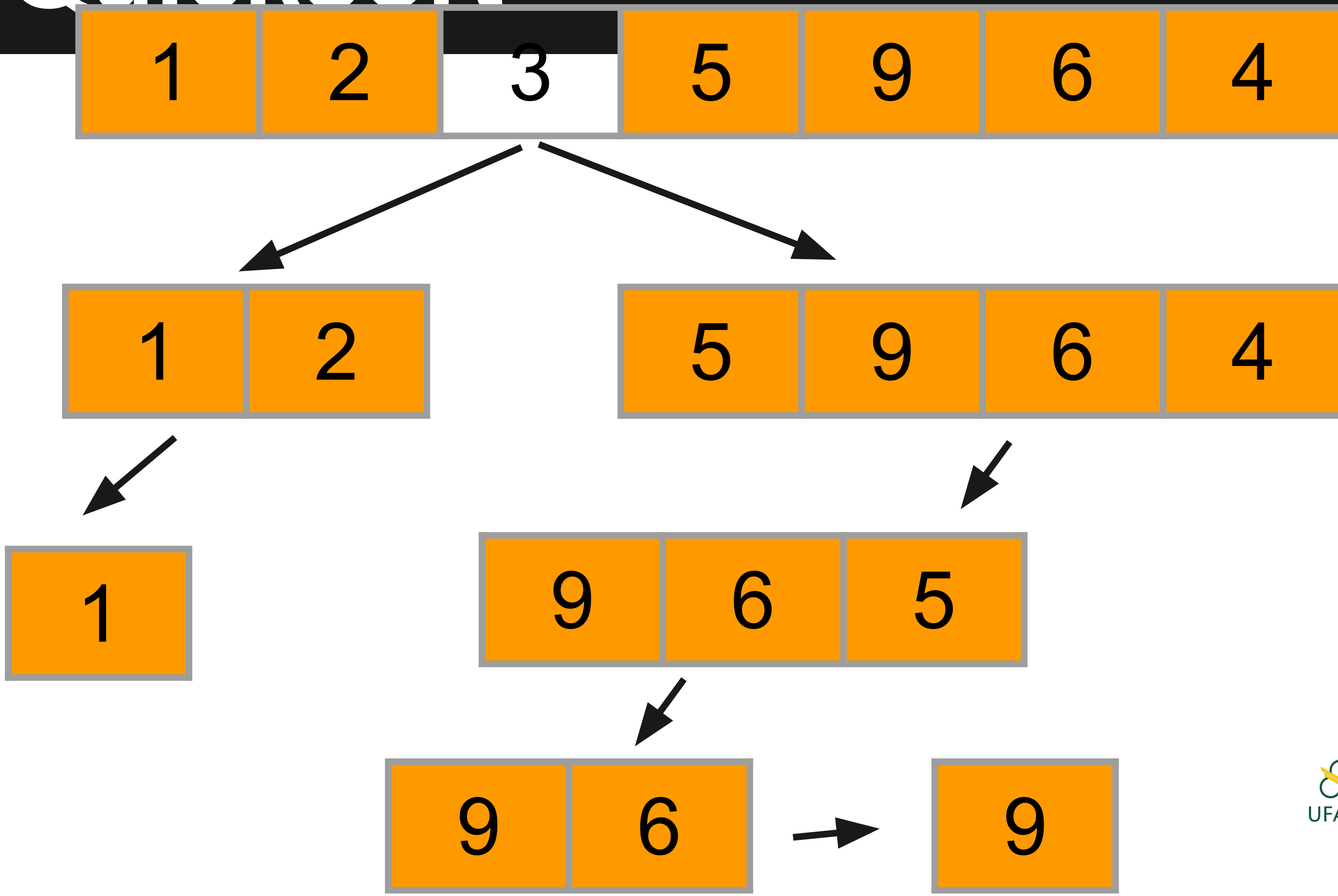
Quicksort

1	2	3	5	9	6	4

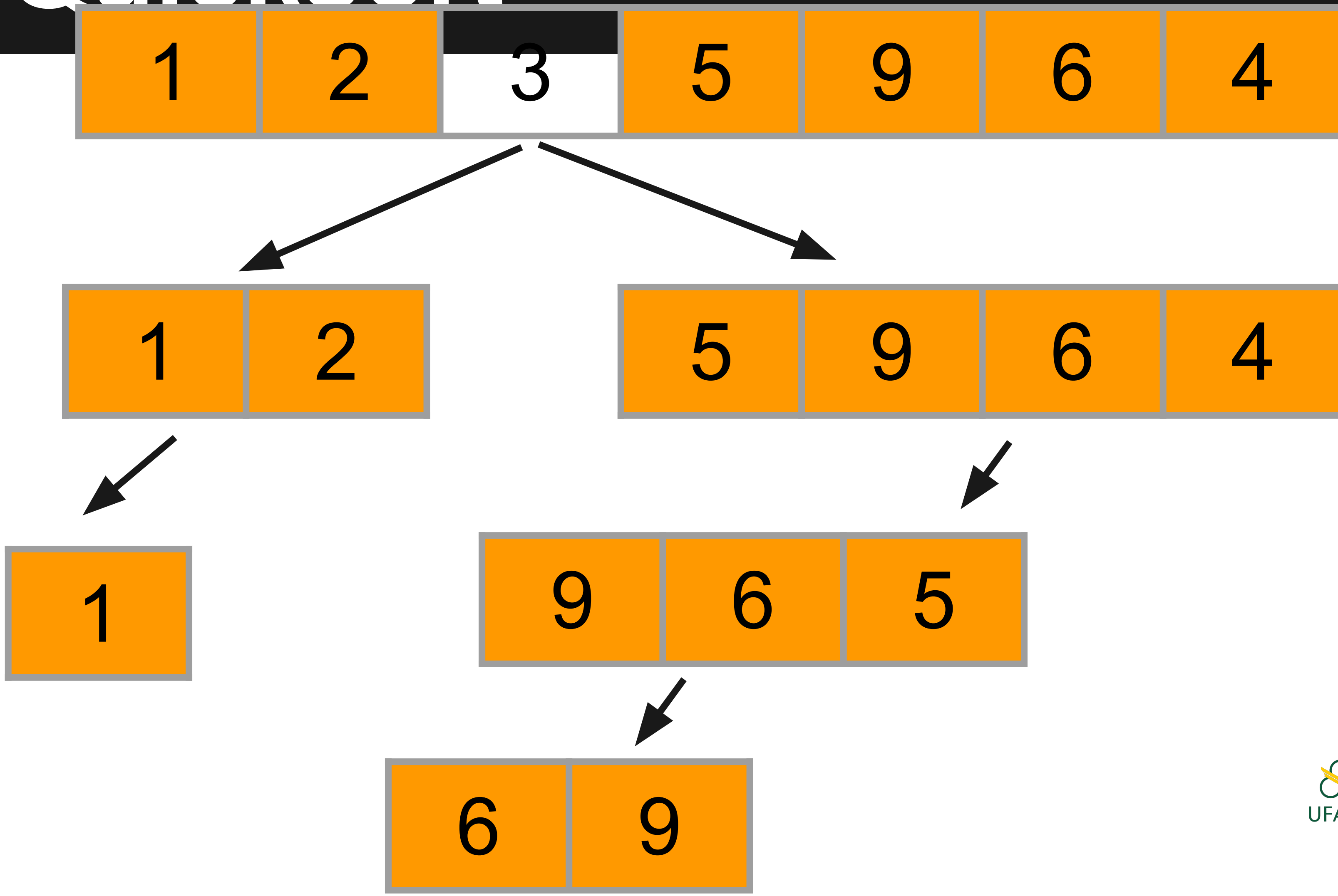
Quicksort



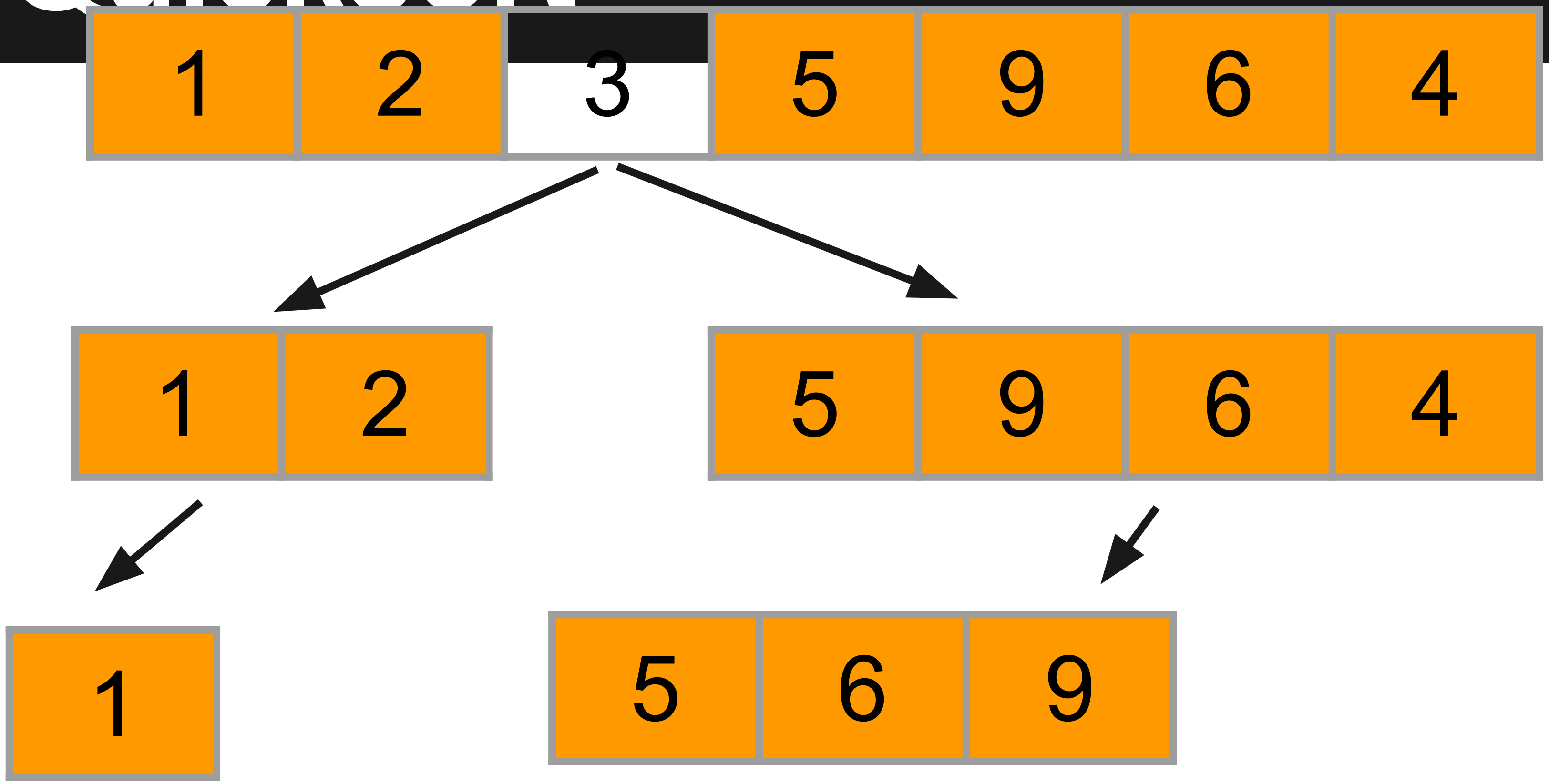
Quicksort



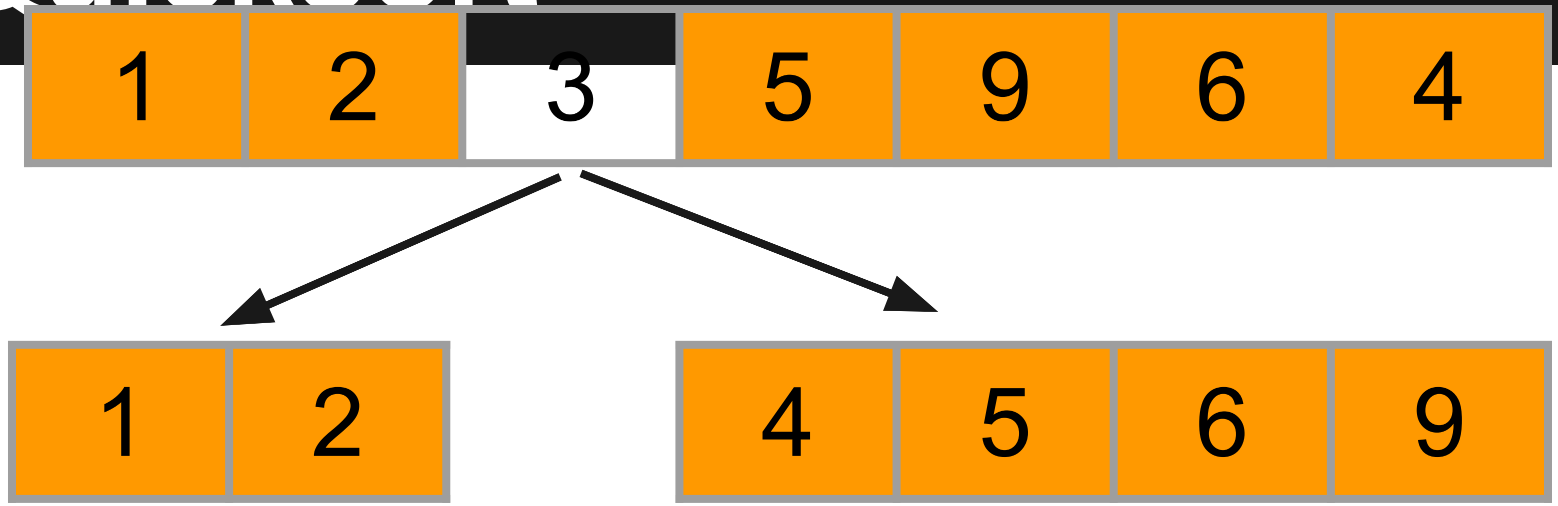
Quicksort



Quicksort



Quicksort



Quicksort

1	2	3	4	5	6	9

UFABC

Quicksort

```
void quicksort (int * v, int low, int high)
{
    int p;
    If (low < high) {
        p = partition(v, low, high);
        quicksort(v, low, p-1);
        quicksort(v, p+1, high);
    }
}
```

Quicksort

```
int partition (int * v, int low, int high)
{
    int pivot = v[high];
    int j, i = low;
    for (j=low; j<high; j++) {
        if (v[j] <= pivot) {
            swap(v, i, j);
            i++;
        }
    }
    swap(v, i, high);
    return i;
}
```

Quicksort

Quantas operações??

A função **partition** executa **high - low** operações. A função principal é chamada recursivamente, divindo a lista em duas em cada recursão.

Quicksort

Se a lista está ordenada, inicialmente executamos **n** operações para particionar e dividimos em resolver o problema para uma lista de tamanho **(n-1)** e uma lista de tamanho **1**.

Esse padrão irá se repetir para cada um dos n-1, totalizando $(n^2 - n)/2$ operações.

UFABC

Quicksort

Isso ocorre por conta da nossa escolha de pivô! Se o pivô cria uma divisão desbalanceada, temos esse número de operações.

UFABC

Quicksort

Por outro lado, se temos uma divisão balanceada, ou seja, o particionamento inicial faz **n** operações, mas divide em duas lista de tamanho **n/2**, temos que:

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

Temos que 2*T(n/2) custa **n** operações. Se as divisões sucessivas forem iguais, teremos **n** . **log n** operações ao todo.

UFABC

Busca

Busca em listas ordenadas

Assumindo uma lista ordenada, podemos tomar certos atalhos na hora de buscar um elemento.

Ao buscar um elemento **x**, podemos iniciar a busca na metade da lista.

UFABC

Busca em listas ordenadas

Quando comparamos x com esse elemento temos 3 situações possíveis:

1. v[metade] == x, ótimo!
2. v[metade] > x
3. v[metade] < x

Nas duas últimas situações podemos dizer, pelo menos, que x está a direita ou a esquerda na lista.

UFABC

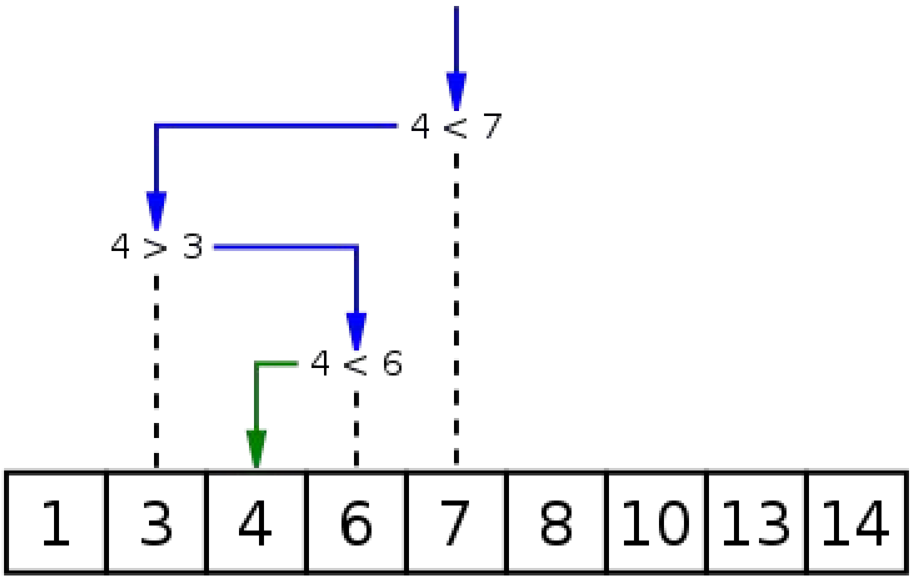
Busca Binária

Porém, sabendo que os elementos estão ordenados podemos fazer o seguinte:

1) Faça h=0 e t=N-1
2) Verifique o elemento m=(h+t)/2
3) Se for maior que x, faça t=m-1
4) Se for menor, faça h=m+1
5) Se for igual, retorna a posição
6) Se h>t, retorna indicando que não encontrou o elemento

UFABC

Busca Binária



https://en.wikipedia.org/wiki/Binary_search_algorithm#/media/File:Binary_search_into_array.png

Busca Binária

A cada passo que alteramos **h** ou **t**, deixamos de olhar para (h+t)/2 elementos tendo a certeza de que o valor **x** não está lá.

No pior caso, quando o elemento não está na lista, repetimos a comparação **log n** vezes.

n = 10, olhamos no 5, 2, 1 → log2(10) ~ 3

Busca por Interpolação

Igual a busca binária, mas assume uma distribuição uniforme dos valores, tenta estimar a posição com:

mid = h + ((x - v[h])*(t-h) / (v[t] - v[h]))

O número de operações estimado é o mesmo, mas tende a ser um pouco menor se a interpolação estiver correta.