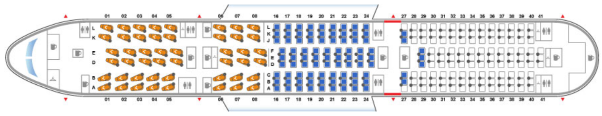
Estruturas de Dados / Programação 2
Algoritmos de Busca

Márcio Ribeiro
marcio@ic.ufal.br
twitter.com/marciomribeiro

Introduction

- Suppose we are using an array to store passengers IDs
- Also, each array index represents a seat



23	12	25	32	33	53	21	2	...	89
0	1	2	3	4	5	6	7	...	540

Systems need to search for customers, products,

- users, passengers, registers, files, accounts etc etc etc!!!

Can you please tell me what is my seat? My ID is 21.

Okay, Sir. Let me look that up for you...

OK! Thanks!

Sir, you're at seat number 6.

So, our systems should search for data!

23	12	25	32	33	53	21	2	...	89
0	1	2	3	4	5	6	7	...	540

Searching algorithms

Linear search

- In this case, we can traverse the array to search for the passenger ID we are interested...
- a.k.a. Sequential search

23	12	25	32	33	53	21	2	...	89
0	1	2	3	4	5	6	7	...	540

Linear search

- Very simple algorithm
- Consists of checking every element, in sequence, until the desired one is found
- Please, take some minutes to write this algorithm…

```
int linear_search(int *v, int size, int element);
```

```
int linear_search(int *v, int size, int element)
{
  int i;
  for (i = 0; i < size; i++) {
    if (v[i] == element) {
      return i;
    }
  }
  return -1;
}
```

Recursive version… in Haskell!

- A little bit different: **true** when found; **false**, otherwise

```
linearSearch :: Int -> [Int] -> Bool
linearSearch e [] = False
linearSearch e (a:as) = if (a == e) then True
                        else linearSearch e as
```

Can you see advantages and disadvantages of this algorithm?

What are the best and worst cases?

Think about it…

- Best case:
  - When the element is at the first position
  - 1 comparison is needed
  - Constant time (it does not depend on the array size!)
- Worst case:
  - When the element is not found
  - n comparisons are needed
  - Time is proportional to the array size

```
for (i = 0; i < size; i++) {
  if (v[i] == element) {
    return i;
  }
}
```

We will come back to this efficiency topic later...

Now, let's see another searching algorithm!

Using the Linear search...

- ... how many comparisons we should do to find 99?

↓	↓	↓	↓	↓		
12	**23**	**45**	**67**	**99**	**101**	**212**
0	1	2	3	4	5	6

```
for (i = 0; i < size; i++) {
  if (v[i] == element) {
    return i;
  }
}
```

Now... instead of saying only "Yes" or "No"...

- Let's imagine some hints like "Left" and "Right"
- Let's try a **non-linear** idea...!
- Where is 99?
  - Let's try position number 5
  - Now, position number 2

		No! →			No! ←	
0	1	2	3	4	5	6

How many times we asked for the element 99?

- We investigated two positions and we know that 99 can be only at two positions: 3 or 4
- That wasn't true before when using the linear search! In this case, we would still need to investigate 5 positions!!!

			↓	↓		
		No! →			No! ←	
0	1	2	3	4	5	6
		↑	↑	↑	↑	↑

Binary search

- By choosing some positions, we narrow down the number of positions we need to take a look!
- Quiz:
  - What is the best position to take a look first? Why?
  - What is the disadvantage of this algorithm?

0	1	2	3	4	5	6

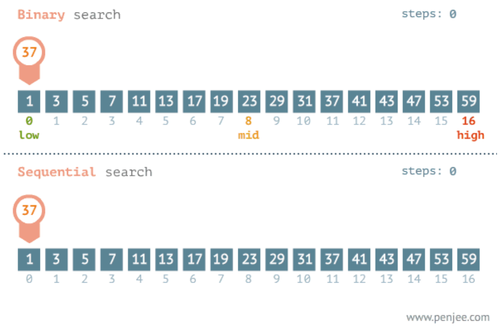
```
int binary_search(int *v, int size, int element);
```

```c
int binary_search(int *v, int size, int element)
{
  int begin = 0;
  int end = size – 1;
  int middle;

  while (begin <= end) {
    middle = (begin + end) / 2;
    if (v[middle] < element) {
      begin = middle + 1;
    } else if (v[middle] > element) {
      end = middle - 1;
    } else {
      return middle;
    }
  }
  return -1;
}
```

Recursive version... in Haskell!

```haskell
import Data.Array;

binarySearch array element begin end
  | begin > end = -1
  | element > array!middle = binarySearch array element (middle + 1) end
  | element < array!middle = binarySearch array element begin (end - 1)
  | otherwise = middle
  where
    middle = (begin + end) `div` 2
```

```
binarySearch (array (0,5) [(0,12),(1,23),(2,45),(3,67),(4,99),(5,101)]) 12 0 5
0
binarySearch (array (0,5) [(0,12),(1,23),(2,45),(3,67),(4,99),(5,101)]) 23 0 5
1
binarySearch (array (0,5) [(0,12),(1,23),(2,45),(3,67),(4,99),(5,101)]) 45 0 5
2
binarySearch (array (0,5) [(0,12),(1,23),(2,45),(3,67),(4,99),(5,101)]) 67 0 5
3
binarySearch (array (0,5) [(0,12),(1,23),(2,45),(3,67),(4,99),(5,101)]) 99 0 5
4
binarySearch (array (0,5) [(0,12),(1,23),(2,45),(3,67),(4,99),(5,101)]) 101 0 5
5
binarySearch (array (0,5) [(0,12),(1,23),(2,45),(3,67),(4,99),(5,101)]) 102 0 5
-1
```



References

Chapter 10