



Análise de algoritmos

# RECURSÃO E ESTRATÉGIA DIVIDIR PARA CONQUISTAR

*Baseado no livro Entendendo Algoritmos, Um guia ilustrado para programadores e outros curiosos. Aditya Y. Bhargava, Novatec, 2017.*





# Análise de algoritmos

# RECURSÃO

Recursão é quando uma função chama a si mesma, como um espelho que reflete a própria imagem, criando uma sequência de chamadas que se repetem até que uma condição seja satisfeita. É utilizado para tornar a resposta mais clara e os *loops* podem apresentar melhora no desempenho do programa.

É importante destacar que existe a probabilidade da recursão acabar em um *loop* infinito e, por isso, a importância em determinarmos uma condição de parada. Em outras palavras, temos:

- **caso-base** é a condição que interrompe a recursão quando o caso base é atingido e;
- **caso recursivo** é quando a função chama a si mesma com um conjunto de dados modificado, se aproximando do caso-base.





# Análise de algoritmos

# **RECURSÃO**

A estrutura de dados que nos ajuda a compreender o funcionamento é a pilha de chamada, utilizada para armazenar múltiplas funções. Cada vez que uma função é chamada, uma caixa de memória é alocada e quando uma função é chamada dentro de outra, a chamada de função fica pausada em estado parcialmente completo.





Análise de algoritmos

# DIVIDIR PARA CONQUISTAR

A estratégia dividir para conquistar é realizar a separação de um grande problema em partes menores e simples de serem resolvidas. Um cenário que exemplifica o conceito de dividir para conquistar é organizarmos livros de uma biblioteca, por meio da:

- **divisão** separação seguindo algum critério como gênero, ordem alfabética por título, ano de publicação ou editora;
- **conquistar** organizar baseado no critério de divisão e;
- **combinação** dos livros.





Análise de algoritmos

# RECURSÃO E DIVIDIR PARA CONQUISTAR

## APLICAÇÕES

A recursão e a estratégia dividir para conquistar são encontrados aplicados aos: algoritmos de ordenação (*merge sort* e *quick sort*), busca binária, algoritmos de grafos (busca de profundidade e largura, algoritmo de Dijkstra), processamento de *strings* (verificação de palíndromos e cálculo de Fibonacci), geometria computacional (divisão e conquista para encontrar o ponto mais próximo), compiladores (análise sintática), inteligência artificial (árvore de decisão) e jogos.

De forma geral, as vantagens da implementação de um algoritmo de recursão fazendo uso da estratégia de dividir para conquistar envolvem a simplicidade, eficiência em termos de tempo de execução, elegância por serem mais legíveis e concisos, modularidade de código.





## Análise de algoritmos

# MERGE SORT E QUICK SORT

Estaremos explorando abaixo, detalhes e implementação sobre os algoritmos mais famosos de ordenação que implementam os conceitos vistos anteriormente.

	MERGE SORT	QUICK SORT
	. . . . .	. . . . .
<b>FUNCIONAMENTO</b>	Divide o <i>array</i> recursivamente em duas metades até cada metade conter somente um elemento, combinando metades ordenadas em uma única lista ordenada.	Escolhe o elemento pivô e particiona o <i>array</i> em elementos menores e maiores que o pivô, ordenando recursivamente as duas partes.
<b>ARMAZENAMENTO</b>	Requer espaço adicional proporcional ao tamanho do <i>array</i> para armazenar sublistas durante a fusão.	Querem na maioria das vezes espaço constante, salvo algumas implementações recursivas que podem usar espaço de pilha.





## Análise de algoritmos

# MERGE SORT E QUICK SORT

	MERGE SORT	QUICK SORT
	. . . . .	. . . . .
FINALIDADE	Estabilidade, ideal para grandes conjuntos de dados, melhor destinados para armazenamento de memória externa.	Mais rápido devido à menor constante, ideal para conjuntos de dados aleatórios e pode ser menos eficiente em conjunto de dados quase ordenados.
COMPLEXIDADE DE TEMPO	$O(n \log n)$ para todos os casos.	$O(n \log n)$ para melhor caso e caso médio, sendo o último tendendo a ser mais rápido por possuir uma constante menor. $O(n^2)$ para pior caso*.

\*caso de *arrays* já ordenados ou quase ordenados, ordenados em ordem inversa e quando todos os elementos do *array* são iguais.





Análise de algoritmos

# MERGE SORT E QUICK SORT

Na sequência, estaremos estudando as implementações do *merge* e *quick sort* para compreendermos as particularidades na prática.

Para o algoritmo de *merge sort*, dada uma lista encadeada, implemente um algoritmo que ordene seus elementos em ordem crescente. A lista encadeada é uma estrutura linear onde cada elemento (nó) contém um valor e um ponteiro para o próximo elemento.

Por outro lado, para algoritmo de *quick sort*, teremos o caso do preenchimento de prateleiras de estantes de forma que seja recebido um *array* contendo espessura e altura do livro, e uma estante com prateleiras de largura fixa. A idéia consiste em organizar esses livros nas prateleiras de forma que a estante fique o mais baixa possível. As regras são:

- Ordem: Os livros devem ser colocados nas prateleiras na mesma ordem em que eles aparecem na lista.







Análise de algoritmos

# MERGE SORT E QUICK SORT

- Largura: A soma das espessuras dos livros em cada prateleira não pode ultrapassar a largura máxima da prateleira.
- Altura: A altura de cada prateleira é determinada pelo livro mais alto daquela prateleira.
- Objetivo: Minimizar a altura total da estante, ou seja, a soma das alturas de todas as prateleiras.





# 🔍 MERGE SORT



```
package main

import (
    "fmt"
    "sort"
)

type LinkedListNode struct {
    Value int
    Next *LinkedListNode
}

type LinkedList struct {
    Head *LinkedListNode
}
```





## 🔍 MERGE SORT



```
func (list *LinkedList) getValues() []int {  
    values := []int{}  
    current := list.Head  
    for current != nil {  
        values = append(values, current.Value)  
        current = current.Next  
    }  
    return values  
}
```

```
func (list *LinkedList) setValues(values []int) {  
    current := list.Head
```





# 🔍 MERGE SORT



```
    for i := range values {  
        current.Value = values[i]  
        if current.Next == nil {  
            break  
        }  
        current = current.Next  
    }  
}
```

```
func (list *LinkedList) Sort() {  
    if list.Head == nil {  
        return  
    }  
}
```





## MERGE SORT



```
values := list.getValues()
sort.Ints(values)
list.setValues(values)
}

func main() {
    firstNode := &LinkedListNode{Value: 5}
    secondNode := &LinkedListNode{Value: -1}
    thirdNode := &LinkedListNode{Value: 30}

    firstNode.Next = secondNode
    secondNode.Next = thirdNode
}
```





## 🔍 MERGE SORT



```
list := &LinkedList{Head: firstNode}
list.Sort()
current := list.Head
for current != nil {
    fmt.Println("the current value of the ordered
linked list is:", current.Value)
    current = current.Next
}
}
```





## QUICK SORT



```
package main

import (
    "fmt"
    "math"
)

type Book struct {
    Thickness int
    Height   int
}
```





## 🔍 QUICK SORT



```
type Shelf struct {  
    Thickness int  
    Height int  
}  
  
func calculateMinimumHeight(books []Book,  
    maximumShelfWidth int) int {  
    if len(books) == 0 {  
        return 0  
    }  
    minimumHeights := make([]int, len(books)+1)
```







## QUICK SORT



```
for i := range minimumHeights {  
    minimumHeights[i] = math.MaxInt32  
}  
minimumHeights[0] = 0  
  
for i := 1; i <= len(books); i++ {  
    currentShelf := Shelf{}  
    for j := i - 1; j >= 0; j-- {  
        if currentShelf.Thickness+books[j].Thickness >  
maximumShelfWidth {  
            break  
        }  
    }  
}
```





## QUICK SORT



```
        currentShelf.Thickness += books[j].Thickness
        currentShelf.Height = max(currentShelf.Height,
books[j].Height)
        minimumHeights[i] = min(minimumHeights[i],
minimumHeights[j]+currentShelf.Height)
    }
}
return minimumHeights[len(books)]
}
```





## 🔍 QUICK SORT



```
func minimumHeightShelves(books [][]int, shelfWidth int)
int {
    bookStruct := make([]Book, len(books))
    for i, book := range books {
        bookStruct[i] = Book{book[0], book[1]}
    }
    return calculateMinimumHeight(bookStruct,
shelfWidth)
}
```





## 🔍 QUICK SORT



```
func main() {  
    books := [][]int{{1, 1}, {2, 3}, {2, 3}, {1, 1}, {1, 1}, {1, 1}, {1,  
2}}  
    shelfWidth := 4  
    minHeight := minimumHeightShelves(books,  
shelfWidth)  
    fmt.Println("the minimum possible height that the total  
bookshelf is:", minHeight)  
}
```





## Q SAÍDA DO PROGRAMA



### **CENÁRIO OTIMISTA *MERGE SORT***

the current value of the ordered linked list is: -1

the current value of the ordered linked list is: 5

the current value of the ordered linked list is: 30

### **CENÁRIO OTIMISTA *QUICK SORT***

the minimum possible height that the total bookshelf is: 6





Análise de algoritmos

# CONSIDERAÇÕES

Para o exemplo apresentado, seguiu-se as boas práticas de desenvolvimento com *clean code* e SOLID, na tentativa de simular um cenário otimista.

Essa iniciativa vai de encontro com a ideia de trazer conteúdos relevantes altamente abordados em processos seletivos e desmitificar a ideia de algoritmos e estrutura de dados. Espero que seja de bom proveito e bons estudos.