
Universidade Federal de São João del-Rei

UNIVERSIDADE FEDERAL DE SÃO JOÃO DEL REI

# Projeto e Análise de Algoritmos

## Trabalho Prático II - Sudoku

**Miguel de Assis Viveiros - 2023008193**
**Nícolas Rafael Mendonça Teles - 2023008200**

Documentação do Trabalho Prático II de Projeto e Análise de Algoritmos. Professor Leonardo Rocha, curso de Ciência da Computação da Universidade Federal de São João Del Rei

**São João Del Rei**
**Dezembro de 2024**

# Sumário

# 1 Introdução

Na Ximguiling Entertainment Games Inc., uma renomada multinacional no setor de jogos eletrônicos, nosso presidente, senhor Michiu, lançou um desafio para motivar e avaliar a equipe. Ele organizou um concurso de algoritmos focado na resolução de probleminhas de Sudoku. Segundo o senhor Michiu, quem desenvolver o algoritmo mais rápido – ou seja, aquele capaz de solucionar um Sudoku no menor tempo possível – receberá um aumento salarial de 300% e será promovido ao cargo de vice-presidente da empresa. Com isso, desenvolvemos duas formas para resolver esse problema, um usando *backtracking* e o outro com uma heurística que criamos, em que resolve primeiro as células com menor número de possibilidades.

## 1.1 Descrição do algoritmo

### 1.1.1 Modo de rodar o programa

O modo esperado de se rodar o programa é o seguinte:

- Primeiro compila-se o programa pelo makefile por meio do comando "make compile"
- Após isso, o executável do programa será gerado. Para rodar o programa em si deve-se seguir a seguinte estrutura:

  - ./main -e [nome_do_arquivo_de_entrada].txt -s [nome_do_arquivo_de_saída].txt -m [modo de rodar o programa](b para *backtracking* ou h para heurística)

- Caso o usuário não informe o modo que deseja que o programa seja rodado, o programa irá resolver de sua maneira padrão, que é o *backtracking*.

### 1.1.2 Lógica do algorimo

Inicialmente, o código recebe pelo terminal o nome do arquivo no qual se encontra a entrada do sudoku e o no qual se deve salvar a saída do mesmo.

Após isso é feita a leitura do arquivo de entrada, no qual lê-se cada célula do sudoku, e caso ela seja um número, o posiciona na matriz e salva na linha, coluna e grid correspondente que aquele número já não pode mais ser utilizado. Caso seja um v na posição, ele preenche um 0 na posição correspondente na matriz

Caso o arquivo de entrada do sudoku não possua uma configuração incorreta - linhas $\neq$ colunas, alguma linha ou coluna irregular em comparação com as outras, ou caso o tamanho do sudoku não seja um quadrado perfeito - ele irá gerar uma estrutura sudoku a partir da entrada, que possui uma minHeap das células vazias - para se utilizar na heurística -; estruturas para salvar os números que ainda podem ser preenchidos em cada linha, coluna e grid; a matriz com o valor presente em cada posição do sudoku; e os

tamanhos necessários para as operações - tamanho do sudoku, de cada grid, e da heap em cada momento.

- *Backtracking*: No backtracking, temos estruturas linha, coluna e grid, cada uma salvando os valores válidos para serem usados na coluna, linha ou grid correspondente, e o sudoku possui um vetor/matriz de cada uma dessas estruturas para que ele consiga representar a grade de sudoku por completa. Dessa maneira não é necessário que se percorra a linha/coluna/grid a cada alteração, pois já temos salvos os valores que podem ser alterados.

  Ao ler o arquivo, o programa já salva por todo o sudoku os valores válidos, para aproveitar essa 'percorrida' no sudoku. Após isso, o código percorre todo o sudoku recursivamente, testando os valores possíveis em cada célula vazia, alterando sua linha/coluna/grid e seguindo para a próxima celula vazia, e caso chegue em alguma célula que não possui nenhum valor válido, volta recursivamente nas células já resolvidas e vai testando seus próximos valores válidos, até que o sudoku esteja resolvido por completo

- Heurística:Da mesma maneira utilizamos as estruturas de linha, coluna e grid, porém também utiliza de uma heap, que organiza as células vazias em ordem crescente, levando em consideração o número de valores possíveis em cada um, heap essa que é preenchida numa percorrida do sudoku logo após estar completo, para que se tenha acesso correto aos valores válidos em cada posição. Os números de valores possíveis das celulas não são modificados uma vez criados na heap, pois para tal teria q se percorrer toda a linha/coluna/grid a cada operação que alterasse o estado do sudoku, e foi decidido que não seria um custo computacional que vale a pena, portanto apesar de haver inserções e remoções que alteram e heap em si, a ordem que o programa irá testar as células não é alterada devido à decisão de não se modificar os números de valores válidos.

  Vale lembrar que apesar de o número de valores válidos na estrutura célula não é alterado, os valores válidos em linhas/colunas/grids ainda são, então durante os testes de valores válidos ainda não são testados valores já utilizados, da mesma maneira que no *backtracking*

  Optamos por esta heurística por ela simular o raciocínio de um jogador ao resolver um Sudoku. Que é priorizar as células com o menor número de possibilidades de valores válidos e as preenchendo primeiro. E a escolha da heap mínima é porque nos permite acessar rapidamente a célula com o menor número de possibilidades com um processo ordenado e otimizado.

# 2 Descrição das Rotinas

## 2.1 Lógica

### 2.1.1 *troca()*

- Descrição: Troca os valores de 2 células, célula a recebe os valores de b, e b de a.
- Parâmetros:

  - a (Tipo Celula*): uma das células a ter seus valores trocados. - b (Tipo Celula*): outra das células a ter seus valores trocados
- Retorno (Tipo void):

  - não é necessário retorno devido ao fato de a função trabalhar com ponteiros.

### 2.1.2 *esquerda()*

- Descrição: Calcula posição do filho esquerdo de i na heap.
- Parâmetros:

  - i (Tipo int): posição da qual se deseja calcular o filho esquerdo
- Retorno (Tipo int):

  - posição do filho esquerdo.

### 2.1.3 *direita()*

- Descrição: Calcula posição do filho direito de i na heap.
- Parâmetros:

  - i (Tipo int): posição da qual se deseja calcular o filho direito
- Retorno (Tipo int):

  - posição do filho direito.

### 2.1.4 *pai()*

- Descrição: Calcula posição do pai de i na heap.
- Parâmetros:

  - i (Tipo int): posição da qual se deseja calcular o pai
- Retorno (Tipo int):

  - posição do pai.

### 2.1.5 *heapify()*

- Descrição: Organiza o vetor da heap em ordem crescente.
- Parâmetros:

  - sudoku (Tipo Sudoku*): ponteiro para a estrutura sudoku.

  - i (Tipo int): posição a partir da qual se deve organizar a heap.
- Retorno (Tipo void):

  - não necessita de retorno por trabalhar somente com referência do Sudoku.

### 2.1.6 *insere()*

- Descrição: Insere uma célula na heap e a reorganiza, onde necessário
- Parâmetros:

  - sudoku (Tipo Sudoku*): ponteiro para a estrutura sudoku.

  - elemento (Tipo celula): célula a ser inserida na heap.
- Retorno (Tipo bool):

  - retorna se foi possível a inserção na heap.

### 2.1.7 *removeMin()*

- Descrição: Remove o menor elemento da heap e reorganiza-a.
- Parâmetros:

  - sudoku (Tipo Sudoku*): ponteiro para a estrutura sudoku.
- Retorno (Tipo Celula):

  - O menor elemento da heap.

### 2.1.8 *validaValor()*

- Descrição: Essa função é responsável por alterar o *boolean*, que é passado por parâmetro, nos vetores validos, linha e coluna, e no grid o valor, que também é passado por parâmetro.
- Parâmetros:

  - s (Tipo Sudoku*): é a estrutura criada para o Sudoku que possui os vetores validos e a matriz de grids validos por valores.

  - i (Tipo int): é o número da linha que deve ser alterada seu *boolean*.

- j (Tipo int): é o número da coluna que deve ser alterada seu *boolean*.

- valor (Tipo int): é o valor do sudoku que terá a validez alterada.

- boolean (Tipo int): é o valor que deseja alterar, se for 0 o valor é inválido naquela linha, coluna e grid, se for 1 o valor é válido para os três.

- Retorno (Tipo void):

- como as mudanças feitas nessa função mexe, somente, com a referência do Sudoku não necessita de retorno.

#### 2.1.9 *quadradoPerfeito()*

- Descrição: Essa função verifica se o tamanho do Sudoku é um quadrado perfeito.

- Parâmetros:

- s (Tipo Sudoku*): é a estrutura criada para o Sudoku que possui o seu tamanho e a raíz quadrada do seu tamanho, que são usadas para a verificação.

- Retorno (Tipo bool):

- retorna 0, ou *false*, se o quadrado da raiz for diferente do tamanho, e 1, ou *true*, se for igual.

#### 2.1.10 *criaSudoku()*

- Descrição: Essa função aloca dinamicamente os componentes da estrutura Sudoku.

- Parâmetros:

- s (Tipo Sudoku*): é a estrutura criada para o Sudoku, que antes já é definido seu tamanho para alocar o restante nessa função.

- Retorno (Tipo void):

- não necessita de retorno por trabalhar somente com referência do Sudoku.

#### 2.1.11 *obterTamanhoSudoku()*

- Descrição: Essa função percorre a primeira linha do Sudoku para obter o seu tamanho e define ele na estrutura.

- Parâmetros:

- s (Tipo Sudoku*): é a estrutura criada para o Sudoku que vai ser definida o seu tamanho e sua raiz nessa função.

- f (Tipo FILE*): é o arquivo de entrada.

- Retorno (Tipo bool):

  - 0 ou false(inválida): caso o tamanho seja 0.

  - 1 ou true(válida): caso seja maior que 0.

#### 2.1.12 *criaVetoresValidos()*

- Descrição: Essa função aloca os vetores, linha, coluna e grid, que verificam se estão validos os valores do Sudoku.

- Parâmetros:

  - s (Tipo Sudoku*): é a estrutura criada para o Sudoku que possui os vetores validos por valores.

- Retorno (Tipo void):

  - não é necessário retorno pois trabalha com a referência do Sudoku.

#### 2.1.13 *preencheValidos()*

- Descrição: Percorre todas as linhas, colunas e grids do sudoku e preenche seus vetores de válidos completamente com 1s.

- Parâmetros:

  - s (Tipo Sudoku*): é a estrutura criada para o Sudoku que possui os vetores validos por valores.

- Retorno (Tipo void):

  - não é necessário retorno pois trabalha com a referência do Sudoku.

#### 2.1.14 *testaValores()*

- Descrição: Percorre os vetores de validos por completo, e sempre que encontrar algum valor válido na linha coluna e grid atual, o insere na posição e chama recursivamente o backtracking para testar se com esse valor nessa posição o sudoku será possível no futuro.

- Parâmetros:

  - s (Tipo Sudoku*): é a estrutura criada para o Sudoku que possui os vetores validos por valores.

  - i (Tipo int): linha na qual deve-se testar o valor.

  - j (Tipo int): coluna na qual deve-se testar o valor.

- Retorno (Tipo int):

  - retorno 0 se foi possível ou 1 se não foi resolver o sudoku no futuro com o valor atual inserido na posição, além disso retorna -1 caso a posição da matriz não seja 0.

#### 2.1.15 *backtracking()*

- Descrição: Função que percorre a matriz a partir do ponto que recebeu, e chama a função testaValores recursivamente nessa posição. A verificação com o j no for é para caso se esteja na linha que recebeu, começar o j a partir do valor recebido, caso não esteja, começar do 0.

- Parâmetros:

  - sudoku (Tipo Sudoku*): ponteiro para a estrutura do sudoku. - n (Tipo int): linha da qual a função deve continuar buscando - m (Tipo int): coluna da qual a função deve continuar buscando

- Retorno (Tipo bool):

  - utiliza o retorno da função testa valores para determinar se o sudoku é possível ou não e retornar esse resultado.

#### 2.1.16 *preencheHeap()*

- Descrição: Função que percorre a matriz do sudoku por completo e caso a posição atual seja 0, verifica quantas possibilidades de números podem ser colocados nela, e salva esse valor.

- Parâmetros:

  - s (Tipo Sudoku*): ponteiro para a estrutura do sudoku.

- Retorno (Tipo void):

  - não é necessário retorno pois trabalha com a referência do Sudoku.

#### 2.1.17 *resolveSudoku()*

- Descrição: Função que recebe o modo que usuário deseja resolver o sudoku e com base nisso decide qual função chamar.

- Parâmetros:

  - sudoku (Tipo Sudoku*): ponteiro para a estrutura do sudoku. - modo (Tipo char)

- Retorno (Tipo bool):

  - Retorno se o sudoku foi possível ou não de se resolver.

#### 2.1.18 *heuristica()*

- Descrição: Função que percorre a heap de células do sudoku, resolvendo primeiramente as com menos possibilidades e caso chegue em um ponto impossível, retorna para suas chamadas anteriores e testa o próximo valor possível na posição.
- Parâmetros:

  - s (Tipo Sudoku*): ponteiro para a estrutura do sudoku.
- Retorno (Tipo bool):

  - Retorno se o sudoku foi possível ou não de se resolver.

#### 2.1.19 *destroiValidos()*

- Descrição: Função que libera a memória alocada para os vetores de válidos em cada coluna, linha e grid.
- Parâmetros:

  - s (Tipo Sudoku*): ponteiro para a estrutura do sudoku.
- Retorno (Tipo void):

  - não é necessário retorno pois apenas libera memória.

#### 2.1.20 *destroiMatriz()*

- Descrição: Função que libera a memória alocada para a matriz do sudoku.
- Parâmetros:

  - s (Tipo Sudoku*): ponteiro para a estrutura do sudoku.
- Retorno (Tipo void):

  - não é necessário retorno pois apenas libera memória.

#### 2.1.21 *destroiSudoku()*

- Descrição: Função que libera a memória alocada para o sudoku, e chama funções que liberam memórias alocadas para partes específicas do sudoku.
- Parâmetros:

  - s (Tipo Sudoku*): ponteiro para a estrutura do sudoku.
- Retorno (Tipo void):

  - não é necessário retorno pois apenas libera memória.

## 2.2 Entrada e Saída de Dados

### 2.2.1 *substituiQuebraDeLinha()*

- Descrição: Substitui o *caracter* de quebra de linha('\n') do final de uma string por um *caracter* de final de string('\0) para evitar erro durante as comparações.

- Parâmetros:

  - string (Tipo char*): string da qual se deseja retirar o *caracter* de quebra de linha.

- Retorno (Tipo void):

  - não retorna nada pois os valores necessários de serem alterados(o \n por \0) são manipulados por ponteiro(char*) dentro da própria função.

### 2.2.2 *obterNomeArquivos()*

- Descrição: Usando o getopt para conseguir o nome dos arquivos que são utilizados no programa.

- Parâmetros:

  - argc (Tipo int): quantidade de argumentos lidos no terminal.

  - argv (Tipo char**): é um vetor de strings e cada string é um argumento.

  - arquivoEntrada (Tipo char**): é um ponteiro para a string do arquivo de entrada com os sudokus que deseja resolver, ou verificar se são possíveis.

  - arquivoSaida (Tipo char**): é um ponteiro para a string do arquivo de saída, onde será escrito as resoluções dos sudokus possíveis.

  - modo (Tipo char*): é um ponteiro de char que vai ser a forma que será resolvido o sudoku, por *backtracking* ou pela heurística.

- Retorno (Tipo void):

  - não retorna nada pois todas as variáveis necessárias são modificadas dentro da função por meio de ponteiros.

### 2.2.3 *printaMatriz()*

- Descrição: Essa função printa no terminal a matriz do sudoku com seus valores.

- Parâmetros:

  - matriz (Tipo int**): é a matriz que será impressa no terminal.

  - tamamho (Tipo int): é o tamanho da matriz, já levando em consideração que possui o mesmo número de linhas e colunas.

- Retorno (Tipo void):

  - não precisa de retorno por causa que a matriz é impressa diretamente no terminal.

#### 2.2.4 *proximoSudoku()*

- Descrição: Essa função serve para caso o sudoku lido for inválido, não ser quadrado perfeito, ter menos linha do que coluna ou uma linha ser maior que a outra, ele muda o ponteiro do arquivo para o próximo sudoku do arquivo de entrada.

- Parâmetros:

  - fp (Tipo FILE*): é o ponteiro que está lendo o arquivo.

- Retorno (Tipo void):

  - não é necessário retorno na função pois as alterações são feitas por meio do ponteiro.

#### 2.2.5 *geraSudoku()*

- Descrição: Essa função serve para, primeiramente, aloca o Sudoku, após isso verifica se ele é quadrado perfeito chamando as funções, *quadradoPerfeito()*, e caso seja é criado Sudoku completo, pela função *criaSudoku*, então é feita a leitura linha por linha e onde estiver 'v' vira 0, e é copiada os valores na matriz do Sudoku e verifica se tem linhas com tamanho diferente. E por fim retorna o Sudoku completo.

- Parâmetros:

  - fp (Tipo FILE*): é o ponteiro que está lendo o arquivo de entrada.

- Retorno (Tipo Sudoku*):

  - retorna o Sudoku alocado e com a matriz do Sudoku completa com seus números, caso a entrada seja válida, se não for então retorna NULL.

#### 2.2.6 *printaResultado()*

- Descrição: Essa função escreve a matriz do sudoku resolvida no arquivo de saída.

- Parâmetros:

  - s (Tipo Sudoku*): é a estrutura do Sudoku que possui a matriz que será escrita no arquivo de saída.

  - fs (Tipo FILE*): é o ponteiro do arquivo de saída.

- Retorno (Tipo void):

  - não precisa de retorno por causa que a matriz é impressa diretamente no arquivo, pelo ponteiro.

# 3 Descrição das Estruturas de Dados

## 3.1 Linha

- Descrição: Essa estrutura de dados serve para verificar se um valor do Sudoku é válido nessa linha.
- Atributos:

  - validos (Tipo int*): é o vetor que será salvo se é válido ou não o valor nessa linha.

## 3.2 Coluna

- Descrição: Essa estrutura de dados serve para verificar se um valor do Sudoku é válido nessa coluna.
- Atributos:

  - validos (Tipo int*): é o vetor que será salvo se é válido ou não o valor nessa coluna.

## 3.3 Grid

- Descrição: Essa estrutura de dados serve para verificar se um valor do Sudoku é válido nesse grid.
- Atributos:

  - validos (Tipo int*): é o vetor que será salvo se é válido ou não o valor nesse grid.

## 3.4 Celula

- Descrição: Estrutura que representa uma célula vazia do sudoku.
- Atributos:

  - i (Tipo int): linha na qual a célula se localiza.

  - j (Tipo int): coluna na qual a célula se localiza.

- $\text{num}_p ossiveis(Tipoint)$ : $quantidadedevalorespossíveisnessaposição$.

## 3.5 Sudoku

- Descrição: Estrutura que representa o sudoku por completo.
- Atributos:

  - tamanho (Tipo int): é o tamanho da linha do Sudoku.

- raizTamanho (Tipo int): é a raiz quadrada do tamanho do Sudoku, é feito em int para que se verifique se é quadrado perfeito.

-linhas (Tipo Linha*): é um vetor de estrutura Linha, que verifica a validez nas linhas do Sudoku.

-colunas (Tipo Coluna*): é um vetor de estrutura Coluna, que verifica a validez nas colunas do Sudoku.

-grids (Tipo Grid**): é uma matriz da estrutura Grid, que verifica a validez nos grids do Sudoku.

-matrizSudoku (Tipo int**): é a matriz onde é preenchida com os valores, para resolver o Sudoku.

-tamHeap (Tipo int): é o tamanho da heap, utilizada somente na heurística.

-heap (Tipo Celula*): é uma heap mínima que guarda na primeira posição a célula com o menor número de possibilidades, somente usada na heurística.

# 4 Análise de complexidade das rotinas

Vamos realizar a análise de complexidade das rotinas do nosso algoritmo, considerando uma operação essencial dentro de um modelo matemático específico para cada rotina, considerando sempre o pior caso do nosso algoritmo.

## 4.1 Lógica

### 4.1.1 *troca()*

Operação essencial dessa função seria a atribuição, a qual é executada 3 vezes, independente do tamanho da entrada, portanto função de complexidade da função é f(n)=3. Utilizando notação O a função é O(1), tempo constante

### 4.1.2 *esquerda()*

Operação essencial dessa função é os cálculos matemáticos, multiplicação e soma, com isso a função seria f(n)=2 operações. Como o a entrada não interfere, a notação O seria O(1).

### 4.1.3 *direita()*

Operação essencial dessa função seria as operações matemáticas, que são uma multiplicação e uma soma, portanto função de complexidade da função é f(n)=2 operações. Utilizando notação O a função é O(1), tempo constante

#### 4.1.4 *pai()*

Operação essencial dessa função seriam as operações matemáticas, que são uma subtração e uma divisão, portanto função de complexidade da função é f(n)=2 operações. Utilizando notação O a função é O(1), tempo constante

#### 4.1.5 *heapify()*

Operação essencial dessa função seria a troca de células na heap podendo-se ignorar as operações com índices. A complexidade de se montar uma heap é conhecidamente f(n)=O(nlog(n)), portando essa é a complexidade dessa função.

#### 4.1.6 *insere()*

Operação essencial dessa função é a troca de célula. A complexidade de inserir um novo elemento na heap já é conhecida e é f(n)=O(log(n)).

#### 4.1.7 *removeMin()*

Operação essencial dessa função é o heapify. Que já tem a complexidade calculada anteriormente que é f(n)=O(nlog(n)).

#### 4.1.8 *validaValor()*

Operação essencial dessa função são 3 atribuições que ocorrem, independentemente do tamanho da entrada, portando f(n)=3, f(n)=O(1) -> tempo constante.

#### 4.1.9 *quadradoPerfeito()*

Operação essencial dessa função é uma comparação, e o número de operações que ocorre não varia com relação ao tamanho da entrada, portando f(n)=1, f(n)=O(1).

#### 4.1.10 *preencheValidos()*

Operação essencial dessa função são atribuições por cada vetor de válidos por cada valor possível no Sudoku que vai ser igual ao tamanho do Sudoku, ou seja, n e em todos os vetores que são n também. Com isso sua complexidade é f(n)= $3n^2$, ou, f(n)= $O(n^2)$.

#### 4.1.11 *criaVetoresValidos()*

Operação essencial dessa função seria a alocação de memória, que acontece 3 vezes para cada $\sqrt{n} * \sqrt{n}$, portando complexidade da f(n)=3n, f(n)=O(n).

#### 4.1.12 *criaSudoku()*

Operação essencial dessa função seriam alocações de memória, que ocorrem 5 vezes indepentende do tamanho, 2 *for-loops* que um aloca n vezes e outro $\sqrt{n}$ vezes, e alem disso chama *criaVetoresValidos* e *preencheValidos*, que foram calculados respectivamente de terem n e $n^2$ por complexidade. Portando a complexidade final da função é: f(n)=$3n^2$+3n+n+$\sqrt{n}$+5 = $3n^2$+4n+$\sqrt{n}$, $f(n) = O(n$š).

#### 4.1.13 *obterTamanhoSudoku()*

Operação essencial dessa função é a incrementação do contador, que é feita pelo tamanho da linha, que será igual ao tamanho do Sudoku, ou seja igual a n. Com isso sua complexidade é f(n)= n, ou, f(n)= O(n).

#### 4.1.14 *testaValores()* e *backtracking()*

Pelo fato das duas funções se chamarem recursivamente uma à outra faremos a análise conjunta das duas. Ao analisar as funções *testaValores()* e *backtracking()* utilizam recursão para resolver o Sudoku. A função *testaValores()* tenta $O(n)$ valores para cada célula, enquanto *backtracking()* percorre todas as $n^2$ células. Devido à natureza recursiva das funções, a complexidade total no pior caso se torna exponencial: $O(n^{n^2})$.

#### 4.1.15 *preeencheHeap()*

A operação essencial dessa função será a verificação dos valores possiveis em cada célula que esteja vazia na matriz, e no pior caso percorrerá $n^2$ e verificar n valores possiveis. Logo a funcão será igual a f(n)=O($n^3$).

#### 4.1.16 *resolveSudoku()*

Devido ao fato da função apenas chamar a função de resolução desejada, no final das contas sua complexidade será a complexidade de uma delas, ambas sendo $O(n^{n^2})$.

#### 4.1.17 *heuristica()*

A função *heuristica()* resolve o Sudoku de maneira recursiva, removendo células do heap e tentando preenchê-las com valores válidos. Apesar de tentar utilizar uma estrátegia para que a resolução do sudoku seja mais rápida, infelizmente a complexidade não é alterada, já que no pior caso, da mesma maneira, teremos que testar todas as possibilidades. Para cada célula, ela testa até $n$ valores possíveis, e há $n$ células no Sudoku. Como a função é recursiva, a complexidade total é $n$, o que é exponencial, semelhante ao *backtracking* tradicional, devido à exploração recursiva de todas as possibilidades.

#### 4.1.18 *destroiValidos()*

A operação considerada será a liberação de memória, que são 3, feitas num loop aninhado que vão até $\sqrt{n}$. Com isso a sua função de complexidade será f(n)=3$\sqrt{n}$*$\sqrt{n}$=3n, em notação O será f(n)=O(n).

#### 4.1.19 *destroiMatriz()*

A operação essencial dessa função será a liberação de memória, que ocorre n vezes no loop, e uma após ele, com isso temos f(n)=n+1, f(n)=O(n).

#### 4.1.20 *destroiSudoku()*

Operação essencial será a liberação de memória, que ocorre 3n em *destroiValidos*, n+1 em *destroiMatriz*, 5 vezes na própria função sem variar de acordo com entrada, e em um loop de $\sqrt{n}$ vezes, portando f(n)=3n+n+1+5+$\sqrt{n}$=4n+$\sqrt{n}$+6, f(n)=O(n)

### 4.2 Entrada e Saída de Dados

#### 4.2.1 *printaMatriz()*

Nessa função consideraremos a impressão no terminal como operação essencial, uma é feita dentro de dois loops aninhados que vão até o tamanho da matriz, ou seja $n^2$, e possui outro após o loop mais interno, que será n. Por fim a sua função será f(n)=$n^2$+n, e em notação O será f(n)=O($n^2$).

#### 4.2.2 *substituiQuebraDeLinha()*

O modelo matemático dessa função definiria a comparação como operação essencial, considerando uma letra e um espaço em média para cada valor do sudoku, a complexidade fica em média 2n, com notação O: O(n).

#### 4.2.3 *obterNomeArquivos()*

Nessa função o modelo matemático definido levará em conta o número de comparações no switch-case. Será feita 6 comparações no total, já que será comparado o -e, o -s e o -m, sendo que no primeiro se for certo será feita somente uma comparação e para os próximos é somado a quantidade de comparação da anterior, então o segundo será 2, e o terceiro 3. Com isso a sua função de complexidade será de f(n) = 6, em notação O será f(n)=O(1), tempo constante.

#### 4.2.4 *próximoSudoku()*

Considerando que o código perceba algo errado com a entrada do sudoku na primeira linha, ele terá que percorrer as próximas n-1 linhas até terminar, assim tendo complexidade O(n).

#### 4.2.5 *geraSudoku()*

Assumindo novamente uma média de um espaço e um caracter para cada entrada no sudoku, podemos pensar em n linhas nessa configuração, assim obtendo complexidade final de n*(2n)=2n², O(n²).

#### 4.2.6 *printaResultado()*

Assumindo mais uma vez a média de um espaço e um caracter por valor no sudoku, teremos a complexidade novamente de n linhas com essa configuração, O(n²).

### 4.3 Main

A operação essencial da main é as chamadas das funções, que já foram calculadas anteriormente, a *obterNomeArquivos()* que é O(1), a *geraSudoku()* que é O(n²), a *printaMatriz()* é O(n²), a *resolveSudoku()* é $O(n^{n^2})$, a *printaResultado()* é O(n²) e por fim a *destroiSudoku()* que é O(n).

Com essas funções de complexidade, temos então, que a complexidade da main será igual a $O(n^{n^2})$. Pois a função *resolveSudoku()* dominará assintoticamente todas as outras funções chamadas.

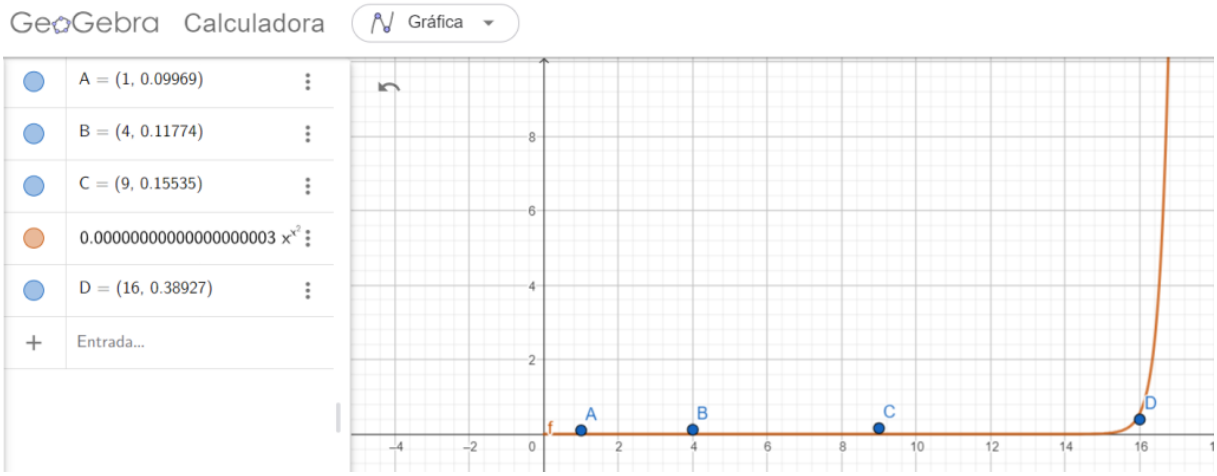
## 5 Análise dos resultados obtidos

Para a análise dos resultados obtidos, usaremos o tempo do processador em um gráfico e outro com o tempo real. E como entrada utilizamos diferentes tamanhos de Sudokus, 1x1, 4x4, 9x9 e 16x16. Ao tentar o de 25x25 esperamos por mais de 11 horas e não obteve resultado devido a complexidade exponencial do problema.

Pelo fato de tanto a função *backtracking()* quanto a *heuristica()* possuírem a mesma complexidade no pior caso, faremos somente uma análise em cada um dos tempos. Então os nossos testes foram pensados no pior caso da heurística, quando todas as entradas forem vazias.

Abaixo está o gráfico com o tempo de processador, sendo essa variável no eixo y, e no eixo x está o tamanho da linha do sudoku. E como foi dito anteriormente as entradas são 1, 4, 9 e 16.


Agora abaixo está o gráfico com o tempo real, com as mesmas entradas utilizadas do anterior.




Nesse gráfico pode-se observar que, conforme variamos o tamanho da linha do Sudoku, o tempo do processador e o tempo real varia na proporção esperada segundo a complexidade calculada - f(n)= $O(n^{n^2})$ - variando de maneira exponencial, e como essa função é com base na quantidade de vezes que é feita a operação, escolhemos uma constante c pequena o suficiente para demonstrá-la nos gráficos que geram o tempo de execução ao invés de número de operações, como a função calcula. Portanto aqui demonstramos a representação real de nossa função de complexidade testada pelo tempo de execução do nosso código.

# 6 Conclusão

Pode-se perceber por meio desse experimento que a resolução de um sudoku trata-se de um dos problemas difíceis da computação, sendo necessário testar todas as suas possibilidades, no pior caso, para se encontrar uma solução. Pode-se ver como problemas de tal classe explodem exponencialmente de uma hora para outra, uma vez que para processar o arquivo de tamanho 16 o programa demora menos que 1 segundo, mas quando

subimos o tamanho para 25 ele ficou horas e horas processando e ainda assim não conseguiu concluir o processo.

Percebe-se também que, apesar de criarmos uma heurística que segue a lógica comum do ser humano para se resolver sudokus - completar as células que têm menos possibilidades primeiro - isso não nos garante uma solução melhor que o *backtracking* puro, uma vez que, como demonstramos com nossos sudokus vazios de exemplo, no pior caso todas as células estão vazias e têm o máximo de possibilidades, e acabamos tendo que testar todas as possibilidades de resolução.