Análise Empírica

Any Ambria da Silva Pinheiro

Centro Universitário FMU (FMU)

São Paulo-SP, Brasil

E-mail: anyaspinheiro@gmail.com

***Resumo* —** **Neste artigo é proposto a análise de alguns algoritmos com o intuito de demonstrar e analisar suas complexidades em situações variadas. Neste artigo, irei analisar os seguintes algoritmos: Bubblesort, Quicksort, jogo Sudoku (*Backtraking)* e desafio do Troco Mínimo (algoritmos gulosos).**

***Palavras-Chaves* — *Análise de Algoritmo, ordenação, Bubblesort, Quicksort, Sudoku, Troco mínimo, Complexidade, Crescimento Assintótico, Vetores;***

1. INTRODUÇÃO

Neste artigo, através de códigos variados e adaptados irei avaliar a complexidade e o tempo de execução de cada algoritmo selecionado. Os códigos foram gerados em linguagem C, utilizando o aplicativo Pelles C e C++, utilizando um compilador online.

O conteúdo deste artigo é estruturado da seguinte forma:

No capítulo II serão apresentados os conceitos referentes aos algoritmos Bubblesort, Quicksort, um código para análise do jogo Sudoku e também um para análise do desafio do troco mínimo. Neste capítulo explorarei um pouco sobre as características, propriedades e principais desafios de cada algoritmo escolhido. No capítulo III será apresentada a metodologia e descrição do processo de adaptação dos códigos para análise da complexidade de cada algoritmo. No capítulo IV será apresentado o resultado dos experimentos com tabelas e gráficos. E no capítulo V serão apresentadas as conclusões tiradas de cada código utilizado.

1. ALGORITMOS ESCOLHIDOS

**Bubblesort:** também conhecido por ordenação por flutuação (literalmente "por bolha"), é um [algoritmo de ordenação](https://pt.wikipedia.org/wiki/Algoritmo_de_ordena%C3%A7%C3%A3o) dos mais simples. A ideia é percorrer o [vetor](https://pt.wikipedia.org/wiki/Vector) diversas vezes, e a cada passagem fazer flutuar para o topo o maior elemento da sequência. Essa movimentação lembra a forma como as bolhas em um [tanque](https://pt.wikipedia.org/wiki/Tanque_(reservat%C3%B3rio)) de água procuram seu próprio nível, e disso vem o nome do algoritmo. Sua complexidade assintótica é **O(N²)**, por causa disso não é recomendado para vetores muito grandes.

**Quicksort:** um método de ordenação muito rápido e eficiente, inventado por Charles Antony Richard Hoare em 1960, quando visitou a [Universidade de Moscovo](https://pt.wikipedia.org/wiki/Universidade_de_Moscovo) como estudante. Naquela época, Hoare trabalhou em um projeto de [tradução de máquina](https://pt.wikipedia.org/w/index.php?title=Tradu%C3%A7%C3%A3o_de_m%C3%A1quina&action=edit&redlink=1) para o [National Physical Laboratory](https://pt.wikipedia.org/w/index.php?title=National_Physical_Laboratory,_UK&action=edit&redlink=1). Ele criou o quicksort ao tentar traduzir um dicionário de inglês para russo, ordenando as palavras, tendo como objetivo reduzir o problema original em subproblemas que possam ser resolvidos mais fácil e rápido. Foi publicado em 1962 após uma série de refinamentos. É o algoritmo de ordenação interna mais rápido que se conhece para uma ampla variedade de situações, utilizando a solução de dividir o problema de ordenar um conjunto com n itens em dois problemas menores, os problemas menores são ordenados independentemente e as partições são combinadas para produzir a solução final. Sua complexidade assintótica **O(n log n).**

**Sudoku:** consiste em verificar através de [lógica matemática](https://pt.wikipedia.org/wiki/L%C3%B3gica_matem%C3%A1tica), as possibilidades para resolver o problema apresentado. Umas das formas de resolver esse problema é o *Backtraking* (Tentativa e erro), no qual consistem em criar uma classe de verificação de inconsistências, chamando-a, a cada número preenchido, se estiver incorreto, reinicia-se com um número diferente, até se obter o número perfeito, técnica bastante trabalhosa, já que para o correto preenchimento, é necessário cerca de um [trilhão](https://pt.wikipedia.org/wiki/Trilh%C3%A3o) de tentativas. Sua complexidade assintótica é **O(N²)**.

**Troco Mínimo:** é um problema clássico da ciência da computação, sendo um dos algoritmos gulosos, que tem como ideia principal retornar o menor número possível de moedas em um troco. Sua complexidade assintótica **O(n)**.

III – METODOLOGIA

Para a análise do algoritmo do bubblesort, foi utilizado o código disponível no site <https://www.geeksforgeeks.org/bubble-sort/> e adaptado com alocação dinâmica de memoria para evitar erro de memoria em vetores muito grandes. Os vetores para teste variaram de 10.000 a 100.000.000, sendo que o 10.000.000 e o 100.000.000 não foram possíveis aferir o tempo, pois o código ficou mais de 12h em execução e mesmo assim não foi concluído.

Já para análise do quicksort foi utilizado o código disponível no site <https://www.geeksforgeeks.org/quick-sort/> e adaptado com alocação dinâmica de memoria para evitar erro de memoria em vetores muito grandes. Os vetores para teste variaram de 10.000 a 100.000.000, e ao contrario do bubblesort, esse código não teve problemas para terminar a execução, por ser um algoritmo extremamente rápido, utilizar um pivô para a divisão e conquista e recursividade.

O Sudoku foi utilizado o código disponível no site <https://www.geeksforgeeks.org/sudoku-backtracking-7/> e adaptado para exibir o resultado do tamanho de cada matriz. Logo após fazer os testes com os tamanhos presentes no gráfico-*Figura 5*, as saídas desse algoritmo são: os segundos de cada operação e a matriz utilizada. Por ultimo cada saída foi adicionada a uma planilha e gráfico a fim de ter uma visão melhor para análise.

Para o troco mínimo, foi utilizado o código do vídeo <https://www.youtube.com/watch?v=SkeqREbtd_k> e alterado nos valores a fim de ser usado de maneira adequada aos experimentos. Logo após fazer os testes com os valores apresentados no gráfico-*Figura 6*, as saídas desse algoritmo são: os microssegundos de cada operação, o valor do troco e quais foram às notas usadas. Por ultimo cada saída foi adicionada a uma planilha e gráfico a fim de ter uma visão melhor para análise.

As linguagens utilizadas foram: C para o desenvolvimento do bubblesort e quicksort, no compilador Pelles C; e C++ para o sudoku e troco mínimo no compilador online.

As configurações da máquina utilizada para executar os testes do bubblesort e quicksort são: Processador Intel(R) Core(TM) i7-357U; Memoria RAM GB; HD 500GB + 24GB de SSD; SO Windows 10. E os teste do sudoku e troco mínimo foram executados no compilador online <https://www.onlinegdb.com/online_c++_compiler> com o navegador Google Chrome, na mesma máquina descrita para o bubblesort.

IV – RESULTADOS

Neste Capítulo, explorarei os resultados obtidos nos testes realizados de cada algoritmo. Todos os testes foram realizados com diferentes valores e testados múltiplas vezes. Em alguns casos, foi calculado também o tempo médio de execução do algoritmo.

Figura 1 - Gráfico do pior caso do bubble sort

*Figura 2 - Gráfico do melhor caso do bubble sort*

Observando os gráficos *Figura 1* e *Figura 2,*  vê-se que o tempo do algoritmo bubble sort segue um crescimento polinomial, compatível com o seu crescimento assintótico O(n²). Com isso nota-se também que não foi possível testar os vetores 10.000.000 e 100.000.000, por causa da sua alta complexidade (os dois vetores citados não finalizaram sua execução mesmo após 12h) e quando comparados aos gráficos do quick sort, o tempo é muito discrepante. Exemplo no bubble sort pior caso no vetor 1.000.000 o tempo é mais que duas mil vezes maior do que o quick sort no pior caso no mesmo vetor.

*Figura 3 - Gráfico do pior caso do quick sort*

*Figura 4 - Gráfico do melhor caso do quick sort*

Observando os gráficos *Figura 3* e *Figura 4,*  vê-se que o tempo do algoritmo quick sort segue um crescimento logaritmo, compatível com o seu crescimento assintótico O(n lg n). Esse algoritmo é extremamente rápido, visto que o mesmo, no maior vetor (100.000.000) no melhor e pior caso, executou em menos de três segundos, o que o algoritmo bubble sort não conseguiu nem terminar de executar. Mostrando assim que o custo do bubble sort é computacionalmente maior que o custo do quick sort.

Interessante também é que se comparar o gráfico *Figura 3* com o *Figura 4,* o tempo são muito semelhantes.

*Figura 5 - Gráfico do Sudoku*

Observando o gráfico *Figura 5*, vê-se o tempo de execução em função do tamanho da matriz e na maioria das vezes vê os tempos muito próximos (como é o caso das matrizes 3, 4, 5 e 8), começando a ficar inclinada (exponencial) a curva na matriz de tamanho 16.

*Figura 6 - Gráfico do Troco Mínimo*

Como pode ser observado no gráfico acima (Figura 6), nem sempre o menor valor de troco terá o menor tempo, e vise versa, isso ocorre porque muitas vezes aquele valor menor terá que passar por um número de verificações maior do quem um número grande, por exemplo, o R$ 201,00 comparado ao R$ 85,00.

V – CONCLUSÃO

 Conforme a demanda por tecnologia aumenta, tem-se a necessidade de armazenar muitos dados hoje em dia, seja em máquinas ou em nuvem (rede), dessa forma, a inserção de dados é imensa. Tendo isso, foram planejados e criados os algoritmos do estilo “Sort”, para estruturar e estabelecer sincrônica e organizacionalmente toda a informação coletada e viabilizada pela ferramenta, programa, ou até terminal.

Vendo a importância dos algoritmos de ordenação, neste artigo pode-se observar o comportamento de quatro algoritmos (dois de ordenação, um de *Backtraking* e um de algoritmo guloso) diante de diversos cenários. No caso dos de ordenação pode-se observar que, evidentemente, quanto maior  vetor, maior será o tempo de execução, porém a divergência entre o tempo de ordenação do BubbleSort com o QuickSort. Isso se da pela complexidade de cada um, o primeiro a codificação é muito mais simples, mas sua complexidade é O(N²) levando muito tempo para ordenar um vetor muito grande. Já o QuickSort a codificação não é tão simples (uma vez que usa recursividade), porém sua complexidade é n lg n, levando menos da metade do tempo que o BubbleSort.

Interessante também é o algoritmo de Troco Mínimo, em que algumas situações como o valor do troco sendo um número grande, o tempo de processamento pode ser pequeno, e o inverso também pode ser verdade, o valor sendo um número pequeno e o tempo de processamento pode ser maior do que um valor alto.

E o Sudoku de forma geral, o tempo de processamento é proporcional ao tamanho da matriz, salvo algumas exceções que uma matriz maior demora menos tempo do que uma menor (como mostra a Figura 5).

REFERÊNCIAS

[1] (s.d.)**.** Acesso em 10 de Maio de 2020, disponível em https://www.geeksforgeeks.org/quick-sort/

[2] (s.d.). Acesso em 2020 de Maio de 2020, disponível em https://www.geeksforgeeks.org/bubble-sort/

[3] (s.d.). Acesso em 15 de Maio de 2020, disponível em https://www.geeksforgeeks.org/sudoku-backtracking-7

[4] (s.d.). Acesso em 15 de Maio de 2020, disponível em https://homepages.dcc.ufmg.br/~cunha/teaching/20121/aeds2/quicksort.pdf

[5] (s.d.). Acesso em 20 de Maio de 2020, disponível em https://www.onlinegdb.com/online\_c++\_compiler

[6] (22 de Março de 2014). Acesso em 20 de Maio de 2020, disponível em http://crbonilha.com/2014/03/22/o-problema-do-troco/

[7] (30 de Setembro de 2019). Acesso em 11 de Maio de 2020, disponível em https://pt.wikipedia.org/wiki/Bubble\_sort

[8] (24 de Janeiro de 2020). Acesso em 11 de Maio de 2020, disponível em https://pt.wikipedia.org/wiki/Quicksort

[9] (16 de Abril de 2020). Acesso em 20 de Maio de 2020, disponível em https://pt.wikipedia.org/wiki/Algoritmo\_de\_sudoku

[10] Albuquerque, G. R. (03 de Maio de 2014). Acesso em 20 de Maio de 2020, disponível em https://pt.slideshare.net/jackocap/anlise-de-algoritmos-algoritmos-gulosos-troco-mnimo

**APENDICE A – CÓDIGO BUBBLE-SORT**

#include <stdio.h>

#include <time.h>

void swap(int \*xp, int \*yp)

{

int temp = \*xp;

\*xp = \*yp;

\*yp = temp;

}

// A function to implement bubble sort

void bubbleSort(int arr[], int n)

{

int i, j;

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

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

if (arr[j] > arr[j+1])

swap(&arr[j], &arr[j+1]);

}

/\* Function to print an array \*/

void printArray(int arr[], int size)

{

int i;

for (i=0; i < size; i++)

printf("%d ", arr[i]);

printf("\n");

}

// Driver program to test above functions

int main()

{

int n = 100000000;

//int arr[t];

int \*arr;

int t = n;

arr = (int \*) malloc(n\*sizeof(int));

for (int i = 0;i < n;i++){

arr[i] = rand() % n;

}

double tini,tfim,tempo;

//sizeof(arr)/sizeof(arr[0]);

//printArray(arr, n);

tini = (double)clock() / CLOCKS\_PER\_SEC;

bubbleSort(arr, n);

tfim = (double)clock() / CLOCKS\_PER\_SEC;

tempo = (double) tfim - tini;

printf("Sorted array: \n");

printf("Tempo: %2.04f s\n\n", tempo);

//printArray(arr, n);

free(arr);

return 0;

}

**APENDICE B – CÓDIGO QUICK-SORT**

#include <stdio.h>

#include <time.h>

#include <stdlib.h>

#define MAX 10000

void quick\_sort(int \*a, int left, int right);

int main(int argc, char\*\* argv)

{

int i;

int t = MAX;

int \*vet;

vet = (int \*) malloc(t\*sizeof(int));

for(i = 0; i < MAX; i++){

//vet[i] = rand()%t; //numeros aleatrios

vet[i] = t;

t--;

}

double tini,tfim,tempo;

tini = (double)clock() / CLOCKS\_PER\_SEC;

quick\_sort(vet, 0, MAX - 1);

tfim = (double)clock() / CLOCKS\_PER\_SEC;

tempo = (double) tfim - tini;

printf("Tempo: %2.04f s\n\n", tempo);

return 0;

}

void quick\_sort(int \*a, int left, int right) {

int i, j, x, y;

i = left;

j = right;

x = a[(left + right) / 2];

while(i <= j) {

while(a[i] < x && i < right) {

i++;

}

while(a[j] > x && j > left) {

j--;

}

if(i <= j) {

y = a[i];

a[i] = a[j];

a[j] = y;

i++;

j--;

}

}

if(j > left) {

quick\_sort(a, left, j);

}

if(i < right) {

quick\_sort(a, i, right);

}

}

**APENDICE A – CÓDIGO SUDOKU**

// A Backtracking program in C to solve Sudoku problem

#include <stdio.h>

// UNASSIGNED is used for empty cells in sudoku grid

#define UNASSIGNED 0

// N is used for the size of Sudoku grid. Size will be NxN

#define N 9

// This function finds an entry in grid that is still unassigned

bool FindUnassignedLocation(int grid[N][N], int &row, int &col);

// Checks whether it will be legal to assign num to the given row, col

bool isSafe(int grid[N][N], int row, int col, int num);

/\* Takes a partially filled-in grid and attempts to assign values to

all unassigned locations in such a way to meet the requirements

for Sudoku solution (non-duplication across rows, columns, and boxes) \*/

bool SolveSudoku(int grid[N][N])

{

int row, col;

// If there is no unassigned location, we are done

if (!FindUnassignedLocation(grid, row, col))

return true; // success!

// consider digits 1 to 9

for (int num = 1; num <= 9; num++)

{

// if looks promising

if (isSafe(grid, row, col, num))

{

// make tentative assignment

grid[row][col] = num;

// return, if success, yay!

if (SolveSudoku(grid))

return true;

// failure, unmake & try again

grid[row][col] = UNASSIGNED;

}

}

return false; // this triggers backtracking

}

/\* Searches the grid to find an entry that is still unassigned. If

found, the reference parameters row, col will be set the location

that is unassigned, and true is returned. If no unassigned entries

remain, false is returned. \*/

bool FindUnassignedLocation(int grid[N][N], int &row, int &col)

{

for (row = 0; row < N; row++)

for (col = 0; col < N; col++)

if (grid[row][col] == UNASSIGNED)

return true;

return false;

}

/\* Returns a boolean which indicates whether an assigned entry

in the specified row matches the given number. \*/

bool UsedInRow(int grid[N][N], int row, int num)

{

for (int col = 0; col < N; col++)

if (grid[row][col] == num)

return true;

return false;

}

/\* Returns a boolean which indicates whether an assigned entry

in the specified column matches the given number. \*/

bool UsedInCol(int grid[N][N], int col, int num)

{

for (int row = 0; row < N; row++)

if (grid[row][col] == num)

return true;

return false;

}

/\* Returns a boolean which indicates whether an assigned entry

within the specified 3x3 box matches the given number. \*/

bool UsedInBox(int grid[N][N], int boxStartRow, int boxStartCol, int num)

{

for (int row = 0; row < 3; row++)

for (int col = 0; col < 3; col++)

if (grid[row+boxStartRow][col+boxStartCol] == num)

return true;

return false;

}

/\* Returns a boolean which indicates whether it will be legal to assign

num to the given row,col location. \*/

bool isSafe(int grid[N][N], int row, int col, int num)

{

/\* Check if 'num' is not already placed in current row,

current column and current 3x3 box \*/

return !UsedInRow(grid, row, num) &&

!UsedInCol(grid, col, num) &&

!UsedInBox(grid, row - row%3 , col - col%3, num)&&

grid[row][col]==UNASSIGNED;

}

/\* A utility function to print grid \*/

void printGrid(int grid[N][N])

{

for (int row = 0; row < N; row++)

{

for (int col = 0; col < N; col++)

printf("%2d", grid[row][col]);

printf("\n");

}

}

/\* Driver Program to test above functions \*/

int main()

{

// 0 means unassigned cells

int grid[N][N] = {{3, 0, 6, 5, 0, 8, 4, 0, 0},

{5, 2, 0, 0, 0, 0, 0, 0, 0},

{0, 8, 7, 0, 0, 0, 0, 3, 1},

{0, 0, 3, 0, 1, 0, 0, 8, 0},

{9, 0, 0, 8, 6, 3, 0, 0, 5},

{0, 5, 0, 0, 9, 0, 6, 0, 0},

{1, 3, 0, 0, 0, 0, 2, 5, 0},

{0, 0, 0, 0, 0, 0, 0, 7, 4},

{0, 0, 5, 2, 0, 6, 3, 0, 0}};

if (SolveSudoku(grid) == true)

printGrid(grid);

else

printf("No solution exists");

return 0;

}

**APENDICE A – CÓDIGO TROCO-MINIMO**

#include <chrono>

#include <iostream>

void troco(float valor){

enum{nota1000,nota500,nota100,nota50,nota20,nota10,nota5,nota2, nota1};

int quant\_notas[9];

quant\_notas[nota1000] = (int)valor/1000;

valor = valor - (quant\_notas[nota1000]\*1000);

quant\_notas[nota500] = (int)valor/500;

valor = valor - (quant\_notas[nota500]\*500);

quant\_notas[nota100] = (int)valor/100;

valor = valor - (quant\_notas[nota100]\*100);

quant\_notas[nota50] = (int)valor/50;

valor = valor - (quant\_notas[nota50]\*50);

quant\_notas[nota20] = (int)valor/20;

valor = valor - (quant\_notas[nota20]\*20);

quant\_notas[nota10] = (int)valor/10;

valor = valor - (quant\_notas[nota10]\*10);

quant\_notas[nota5] = (int)valor/5;

valor = valor - (quant\_notas[nota5]\*5);

quant\_notas[nota2] = (int)valor/2;

valor = valor - (quant\_notas[nota2]\*2);

quant\_notas[nota1] = (int)valor/1;

valor = valor - (quant\_notas[nota1]\*1);

std::cout<<"\nNotas de 1000,00: "<<quant\_notas[nota1000];

std::cout<<"\nNotas de 500,00: "<<quant\_notas[nota500];

std::cout<<"\nNotas de 100,00: "<<quant\_notas[nota100];

std::cout<<"\nNotas de 50,00: "<<quant\_notas[nota50];

std::cout<<"\nNotas de 20,00: "<<quant\_notas[nota20];

std::cout<<"\nNotas de 10,00: "<<quant\_notas[nota10];

std::cout<<"\nNotas de 5,00: "<<quant\_notas[nota5];

std::cout<<"\nNotas de 2,00: "<<quant\_notas[nota2];

std::cout<<"\nNotas de 1,00: "<<quant\_notas[nota1];

}

int main(){

float valor;

std::cout << "Digite um valor: " << std::endl;

std::cin>>valor;

auto inicio = std::chrono::high\_resolution\_clock::now();

troco(valor);

auto resultado = std::chrono::high\_resolution\_clock::now() - inicio;

long long tempo = std::chrono::duration\_cast<std::chrono::microseconds>(resultado).count();

std::cout<<"\ntempo: "<<tempo;

return 0;

}