Pontifícia Universidade Católica de Minas Gerais

***Campus* Poços de Caldas**

Curso de Ciência da Computação

Rodrigo Franco de Melo Nogueira

Trabalho de análise de algoritmos de busca e de ordenação

MINAS GERAIS

2023/2

Rodrigo Franco de Melo Nogueira

Trabalho de análise de algoritmos de busca e de ordenação

Trabalho Final do sétimo período apresentado ao Campus Poços de Caldas da Pontifícia Universidade Católica de Minas Gerais.

Professor: João Carlos Morselli Junior

MINAS GERAIS

2023/2

1 OBJETIVO E DESCRIÇÃO DOS ALGORITMOS

1.1 BogoSort

O Bogo Sort é um algoritmo de ordenação que possui um comportamento altamente ineficiente e é usado mais como uma piada ou desafio do que como uma solução prática para ordenação. O objetivo do Bogo Sort é simplesmente embaralhar aleatoriamente os elementos de uma lista e verificar se a lista está ordenada. Se não estiver, o algoritmo repete o processo até que a lista seja ordenada.

Objetivo: O objetivo do BogoSort é ordenar uma lista de elementos, mas ao contrário de outros algoritmos de ordenação que seguem estratégias eficientes, o BogoSort simplesmente gera permutações aleatórias até encontrar uma que esteja ordenada.

Complexidade: A complexidade do BogoSort é extremamente alta e não é prática para conjuntos de dados de tamanho significativo. A complexidade de tempo é, em média, O((n+1)!) - fatorial do tamanho da lista mais um. Isso significa que o BogoSort pode exigir uma quantidade exponencial de tempo para ordenar uma lista.

Utilização: O BogoSort não é utilizado na prática para ordenação devido à sua extrema ineficiência. Sua principal utilização é em contextos humorísticos, como uma piada sobre algoritmos de ordenação ineficientes. Algumas pessoas usam o BogoSort como um desafio de programação, pedindo aos outros para implementá-lo e observar quanto tempo leva para ordenar uma lista de tamanho razoável.

1.2 ShellSort

ShellSort é um algoritmo de ordenação que melhora a eficiência do algoritmo de ordenação por inserção, especialmente para conjuntos de dados de grande tamanho. Foi proposto por Donald Shell em 1959. O objetivo principal do ShellSort é reduzir a quantidade de trocas necessárias em comparação com o algoritmo de ordenação por inserção.

Objetivo: O ShellSort visa superar uma das principais desvantagens do algoritmo de ordenação por inserção. Em vez de comparar elementos adjacentes, o ShellSort compara elementos separados por um intervalo fixo chamado "intervalo de salto" (gap). Ele reduz a quantidade de trocas necessárias para colocar elementos em suas posições finais, aumentando o gap gradualmente e realizando inserções sort em subconjuntos dos dados.

Complexidade: A complexidade do Shell Sort varia dependendo da sequência de intervalos escolhida. Em geral, é difícil determinar uma complexidade precisa, pois ela depende do comportamento específico do conjunto de dados. No entanto, em média, o ShellSort tem uma complexidade de pior caso entre O(n log^2 n) e O(n^2), tornando-se mais eficiente do que o algoritmo de ordenação por inserção em muitos casos.

Utilização: O ShellSort é utilizado quando a eficiência do algoritmo de ordenação por inserção é desejada, mas o tamanho do conjunto de dados é grande. Ele é particularmente eficaz para conjuntos de dados parcialmente ordenados. Embora existam outras abordagens mais avançadas, como algoritmos de ordenação baseados em comparação, o ShellSort pode ser uma escolha razoável para conjuntos de dados de tamanho moderado em situações onde a simplicidade de implementação é importante.

1.3 Busca por Jump (Jump Search)

Jump Search é um algoritmo de busca em um conjunto de dados ordenado. O objetivo principal do Jump Search é encontrar a posição de um elemento em uma lista ordenada, pulando blocos fixos em vez de percorrer cada elemento individualmente.

Objetivo: O principal objetivo do Jump Search é reduzir o número de comparações necessárias para encontrar um elemento em um conjunto de dados ordenado. Ao pular blocos fixos, o algoritmo consegue reduzir o número de comparações em comparação com uma busca linear tradicional. Isso faz com que o Jump Search seja eficiente para grandes conjuntos de dados.

Funcionamento: Divide o array em blocos de tamanho $\sqrt{n}$ e realiza uma busca linear no bloco onde o elemento pode estar.

Complexidade: $O(\sqrt{n})$ no melhor e no caso médio, e O(n) no pior caso.

Utilização: O Jump Search é útil quando se trabalha com grandes conjuntos de dados ordenados. Ele é especialmente eficiente quando é possível estimar aproximadamente onde o elemento desejado pode estar. O Jump Search é utilizado em situações em que uma busca binária não é prática devido à impossibilidade de acessar elementos arbitrários, como em listas vinculadas.

1.4 Busca por Interpolação (Interpolation Search)

Busca por Interpolação é um algoritmo de busca em um conjunto de dados ordenado. O objetivo principal da Busca por Interpolação é encontrar a posição de um elemento em uma lista ordenada, ajustando a estimativa da posição com base na distribuição dos valores no conjunto de dados.

Objetivo: O objetivo da Busca por Interpolação é melhorar a eficiência da busca em conjuntos de dados uniformemente distribuídos. Em vez de usar um salto fixo, como em algoritmos de busca como a busca binária, a Busca por Interpolação utiliza uma estimativa interpolada para calcular a posição do elemento desejado. Isso é particularmente eficaz quando o conjunto de dados tem uma distribuição uniforme.

Funcionamento: Estima a posição do elemento de busca usando a fórmula de interpolação e realiza a busca nessa área.

Complexidade: O(log log n) no melhor caso, O(n) no pior caso. O desempenho depende da distribuição dos dados.

Utilização: A Busca por Interpolação é adequada para conjuntos de dados que possuem uma distribuição uniforme dos valores. É importante notar que, em conjuntos de dados não uniformemente distribuídos, a Busca por Interpolação pode não oferecer benefícios significativos em relação a outros algoritmos de busca, como a busca binária.

2 CONCEITOS UTILIZADOS

2.1 BogoSort/ ShellSort: Complexidade de Tempo:

Mede o tempo de execução do algoritmo em função do tamanho da entrada.

Casos de Análise: Pior caso, melhor caso e caso médio.

Funções de Cálculo: Análise matemática para determinar a complexidade de tempo.

2.2 Busca por Jump/ Busca por Interpolação:

Complexidade de Tempo: Determina a eficiência do algoritmo com base no tamanho da entrada.

Testes de Desempenho: Medição do tempo real de execução do algoritmo para diferentes tamanhos de entrada.

3 IMPLEMENTAÇÃO DE ALGORITMOS

Para cada algoritmo feito, foi inserido um Array dinâmico de até 100 espaços começando do tamanho cinco, e a cada execução aumentaria um, pensando nesse caso foi inserido um código para verificar o tempo de execução de cada tamanho de array, foi inserido o array antes e depois da ordenação e logo abaixo o tempo de execução.

3.1 BogoSort

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void shellsort(int arr[], int n) {
    for (int gap = n / 2; gap > 0; gap /= 2) {
        for (int i = gap; i < n; i += 1) {
            int temp = arr[i];
            int j;
            for (j = i; j >= gap && arr[j - gap] > temp; j -= gap) {
                arr[j] = arr[j - gap];
            }
            arr[j] = temp;
        }
    }
}

void print_array(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}
```

```
void generate_best_case(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        arr[i] = i;  // Melhor caso: array já ordenado
    }
}

void generate_worst_case(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        arr[i] = size - i - 1;  // Pior caso: array em ordem inversa
    }
}

void clear_file(const char *filename) {
    FILE *file = fopen(filename, "w");
    if (file == NULL) {
        fprintf(stderr, "Falha ao abrir o arquivo %s.\n", filename);
        exit(1);
    }
    fclose(file);
}

int main() {
    srand(time(NULL));
    int initial_size = 5;

    // Limpar os arquivos antes de começar
    clear_file("melhorcaso.txt");
    clear_file("piorcaso.txt");

    for (int size = initial_size; size <= 1000; size++) {
        int *arr = realloc(NULL, size * sizeof(int));
```

```
if (arr == NULL) {
    fprintf(stderr, "Falha na alocação de memória.\n");
    return 1;
}

// Melhor caso
generate_best_case(arr, size);

printf("Melhor Caso - Array original:\n");
print_array(arr, size);

clock_t start_time = clock();
shellsort(arr, size);
clock_t end_time = clock();

double time_spent = (double)(end_time - start_time) / CLOCKS_PER_SEC;
printf("Melhor Caso - Array ordenado:\n");
print_array(arr, size);
printf("Tempo para ordenar: %.6f segundos\n\n", time_spent);

// Salvar resultados no arquivo
FILE *best_case_file = fopen("melhorcaso.txt", "a");
fprintf(best_case_file, "%d %.6f\n", size, time_spent);
fclose(best_case_file);

free(arr);

// Pior caso
arr = realloc(NULL, size * sizeof(int));
generate_worst_case(arr, size);
```

```
        printf("Pior Caso - Array original:\n");
        print_array(arr, size);

        start_time = clock();
        shellsort(arr, size);
        end_time = clock();

        time_spent = (double)(end_time - start_time) / CLOCKS_PER_SEC;
        printf("Pior Caso - Array ordenado:\n");
        print_array(arr, size);
        printf("Tempo para ordenar: %.6f segundos\n\n", time_spent);

        // Salvar resultados no arquivo
        FILE *worst_case_file = fopen("piorcaso.txt", "a");
        fprintf(worst_case_file, "%d %.6f\n", size, time_spent);
        fclose(worst_case_file);

        free(arr);
    }

    // Executar script Python
    system("python script.py");

    return 0;
}
```

3.2 ShellSort

```
#include <stdio.h>
#include <stdlib.h>
```

```c
#include <time.h>

void shellsort(int arr[], int n) {
    for (int gap = n / 2; gap > 0; gap /= 2) {
        for (int i = gap; i < n; i += 1) {
            int temp = arr[i];
            int j;
            for (j = i; j >= gap && arr[j - gap] > temp; j -= gap) {
                arr[j] = arr[j - gap];
            }
            arr[j] = temp;
        }
    }
}

void print_array(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

void generate_best_case(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        arr[i] = i;  // Melhor caso: array já ordenado
    }
}

void generate_worst_case(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        arr[i] = size - i - 1;  // Pior caso: array em ordem inversa
```

```
    }
}

void clear_file(const char *filename) {
    FILE *file = fopen(filename, "w");
    if (file == NULL) {
        fprintf(stderr, "Falha ao abrir o arquivo %s.\n", filename);
        exit(1);
    }
    fclose(file);
}

int main() {
    srand(time(NULL));
    int initial_size = 5;

    // Limpar os arquivos antes de começar
    clear_file("melhorcaso.txt");
    clear_file("piorcaso.txt");

    for (int size = initial_size; size <= 1000; size++) {
        int *arr = realloc(NULL, size * sizeof(int));

        if (arr == NULL) {
            fprintf(stderr, "Falha na alocação de memória.\n");
            return 1;
        }

        // Melhor caso
        generate_best_case(arr, size);
```

```
printf("Melhor Caso - Array original:\n");
print_array(arr, size);

clock_t start_time = clock();
shellsort(arr, size);
clock_t end_time = clock();

double time_spent = (double)(end_time - start_time) / CLOCKS_PER_SEC;
printf("Melhor Caso - Array ordenado:\n");
print_array(arr, size);
printf("Tempo para ordenar: %.6f segundos\n\n", time_spent);

// Salvar resultados no arquivo
FILE *best_case_file = fopen("melhorcaso.txt", "a");
fprintf(best_case_file, "%d %.6f\n", size, time_spent);
fclose(best_case_file);

free(arr);

// Pior caso
arr = realloc(NULL, size * sizeof(int));
generate_worst_case(arr, size);

printf("Pior Caso - Array original:\n");
print_array(arr, size);

start_time = clock();
shellsort(arr, size);
end_time = clock();

time_spent = (double)(end_time - start_time) / CLOCKS_PER_SEC;
```

```
        printf("Pior Caso - Array ordenado:\n");
        print_array(arr, size);
        printf("Tempo para ordenar: %.6f segundos\n\n", time_spent);

        // Salvar resultados no arquivo
        FILE *worst_case_file = fopen("piorcaso.txt", "a");
        fprintf(worst_case_file, "%d %.6f\n", size, time_spent);
        fclose(worst_case_file);

        free(arr);
    }

    // Executar script Python
    system("python script.py");

    return 0;
}
```

3.3 Busca por Jump

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

int jump_search(int arr[], int n, int x) {
    int step = sqrt(n);
    int prev = 0;
    while (arr[min(step, n) - 1] < x) {
        prev = step;
        step += sqrt(n);
```

```
        if (prev >= n) {
            return -1;
        }
    }
    while (arr[prev] < x) {
        prev++;
        if (prev == min(step, n)) {
            return -1;
        }
    }
    if (arr[prev] == x) {
        return prev;
    }
    return -1;
}

int min(int a, int b) {
    return (a < b) ? a : b;
}

void print_array(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

void generate_best_case(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        arr[i] = i;  // Melhor caso: array já ordenado
    }
```

```
}

void generate_worst_case(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        arr[i] = size - i - 1;  // Pior caso: array em ordem inversa
    }
}

void clear_file(const char *filename) {
    FILE *file = fopen(filename, "w");
    if (file == NULL) {
        fprintf(stderr, "Falha ao abrir o arquivo %s.\n", filename);
        exit(1);
    }
    fclose(file);
}

int main() {
    srand(time(NULL));
    int initial_size = 5;

    // Limpar os arquivos antes de começar
    clear_file("melhorcaso.txt");
    clear_file("piorcaso.txt");

    for (int size = initial_size; size <= 1000; size++) {
        int *arr = realloc(NULL, size * sizeof(int));

        if (arr == NULL) {
            fprintf(stderr, "Falha na alocação de memória.\n");
            return 1;
```

```
}

// Melhor caso
generate_best_case(arr, size);

printf("Melhor Caso - Array:\n");
print_array(arr, size);

int x = size - 1;  // Último elemento

clock_t start_time = clock();
int result = jump_search(arr, size, x);
clock_t end_time = clock();

double time_spent = (double)(end_time - start_time) / CLOCKS_PER_SEC;
printf("Melhor Caso - Elemento %d encontrado na posição %d.\n", x, result);
printf("Melhor Caso - Tempo para buscar: %.6f segundos\n\n", time_spent);

// Salvar resultados no arquivo
FILE *best_case_file = fopen("melhorcaso.txt", "a");
fprintf(best_case_file, "%d %.6f\n", size, time_spent);
fclose(best_case_file);

free(arr);

// Pior caso
arr = realloc(NULL, size * sizeof(int));
generate_worst_case(arr, size);

printf("Pior Caso - Array:\n");
print_array(arr, size);
```

```
        start_time = clock();
        result = jump_search(arr, size, x);
        end_time = clock();

        time_spent = (double)(end_time - start_time) / CLOCKS_PER_SEC;
        printf("Pior Caso - Elemento %d encontrado na posição %d.\n", x, result);
        printf("Pior Caso - Tempo para buscar: %.6f segundos\n\n", time_spent);

        // Salvar resultados no arquivo
        FILE *worst_case_file = fopen("piorcaso.txt", "a");
        fprintf(worst_case_file, "%d %.6f\n", size, time_spent);
        fclose(worst_case_file);

        free(arr);
    }

    // Executar script Python
    system("python script.py");

    return 0;
}
```

3.4 Busca por Interpolação

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int interpolation_search(int arr[], int n, int x) {
```

```
    int lo = 0, hi = (n - 1);
    while (lo <= hi && x >= arr[lo] && x <= arr[hi]) {
        if (lo == hi) {
            if (arr[lo] == x) return lo;
            return -1;
        }
        int pos = lo + (((double)(hi - lo) / (arr[hi] - arr[lo])) * (x - arr[lo]));
        if (arr[pos] == x) {
            return pos;
        }
        if (arr[pos] < x) {
            lo = pos + 1;
        } else {
            hi = pos - 1;
        }
    }
    return -1;
}

void print_array(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

void generate_best_case(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        arr[i] = i;  // Melhor caso: array já ordenado
    }
}
```

```
void generate_worst_case(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        arr[i] = size - i - 1;  // Pior caso: array em ordem inversa
    }
}

void clear_file(const char *filename) {
    FILE *file = fopen(filename, "w");
    if (file == NULL) {
        fprintf(stderr, "Falha ao abrir o arquivo %s.\n", filename);
        exit(1);
    }
    fclose(file);
}

int main() {
    srand(time(NULL));
    int initial_size = 5;

    // Limpar os arquivos antes de começar
    clear_file("melhorcaso.txt");
    clear_file("piorcaso.txt");

    for (int size = initial_size; size <= 3500; size++) {
        int *arr = realloc(NULL, size * sizeof(int));

        if (arr == NULL) {
            fprintf(stderr, "Falha na alocacao de memoria.\n");
            return 1;
        }
```

```
// Melhor caso
generate_best_case(arr, size);

printf("Melhor Caso - Array:\n");
print_array(arr, size);

int x = size - 1;  // Último elemento

clock_t start_time = clock();
int result = interpolation_search(arr, size, x);
clock_t end_time = clock();

double time_spent = (double)(end_time - start_time) / CLOCKS_PER_SEC;
printf("Melhor Caso - Elemento %d encontrado na posicao %d.\n", x, result);
printf("Melhor Caso - Tempo para buscar: %.6f segundos\n\n", time_spent);

// Salvar resultados no arquivo
FILE *best_case_file = fopen("melhorcaso.txt", "a");
fprintf(best_case_file, "%d %.6f\n", size, time_spent);
fclose(best_case_file);

free(arr);

// Pior caso
arr = realloc(NULL, size * sizeof(int));
generate_worst_case(arr, size);

printf("Pior Caso - Array:\n");
print_array(arr, size);
```

```
        start_time = clock();
        result = interpolation_search(arr, size, x);
        end_time = clock();

        time_spent = (double)(end_time - start_time) / CLOCKS_PER_SEC;
        printf("Pior Caso - Elemento %d encontrado na posicao %d.\n", x, result);
        printf("Pior Caso - Tempo para buscar: %.6f segundos\n\n", time_spent);

        // Salvar resultados no arquivo
        FILE *worst_case_file = fopen("piorcaso.txt", "a");
        fprintf(worst_case_file, "%d %.6f\n", size, time_spent);
        fclose(worst_case_file);

        free(arr);
    }

    // Executar script Python
    system("script.py");

    return 0;
}
```

4 RESULTADOS E ANÁLISE DOS TESTE DE DESEMPENHO

Os testes de desempenho foram feitas por um código para calcular tempo das funções, segue código abaixo:

```
import os
import matplotlib.pyplot as plt

def read_file(filename):
```

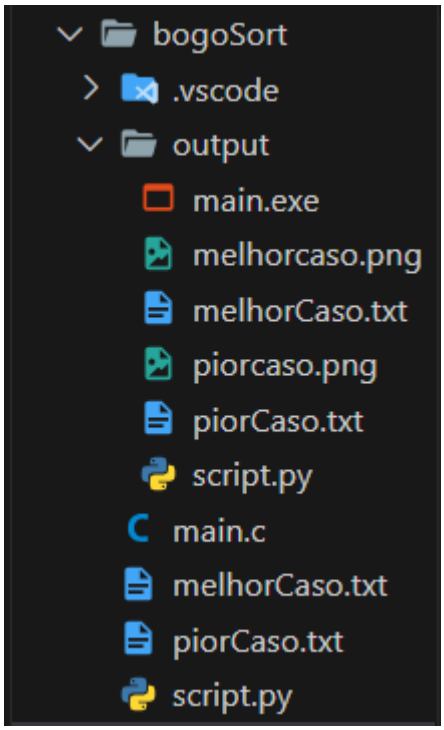
```
    sizes = []
    times = []
    with open(filename, 'r') as file:
        for line in file:
            size, time = map(float, line.split())
            sizes.append(size)
            times.append(time)
    return sizes, times

def plot_and_save(filename, sizes, times, label):
    plt.plot(sizes, times, label=label)
    plt.xlabel('Tamanho do Array')
    plt.ylabel('Tempo (s)')
    plt.title(f'Desempenho do Bogosort - {label}')
    plt.legend()
    script_dir = os.path.dirname(os.path.abspath(__file__))
    image_path = os.path.join(script_dir, filename)
    plt.savefig(image_path)
    plt.close()

# Leitura dos arquivos
melhorcaso_sizes, melhorcaso_times = read_file('melhorcaso.txt')
pirocaso_sizes, pirocaso_times = read_file('pirocaso.txt')

# Plotagem e salvamento das imagens
plot_and_save('melhorcaso.png', melhorcaso_sizes, melhorcaso_times, 'Melhor Caso')
plot_and_save('pirocaso.png', pirocaso_sizes, pirocaso_times, 'Pior Caso')
```

A cada execução o algoritmo irá criar um gráfico novo e inserir como imagem no diretório como mostrado abaixo:



4.1 BogoSort

Os testes foram realizados com tamanhos de lista muito pequenos devido à ineficiência do algoritmo.

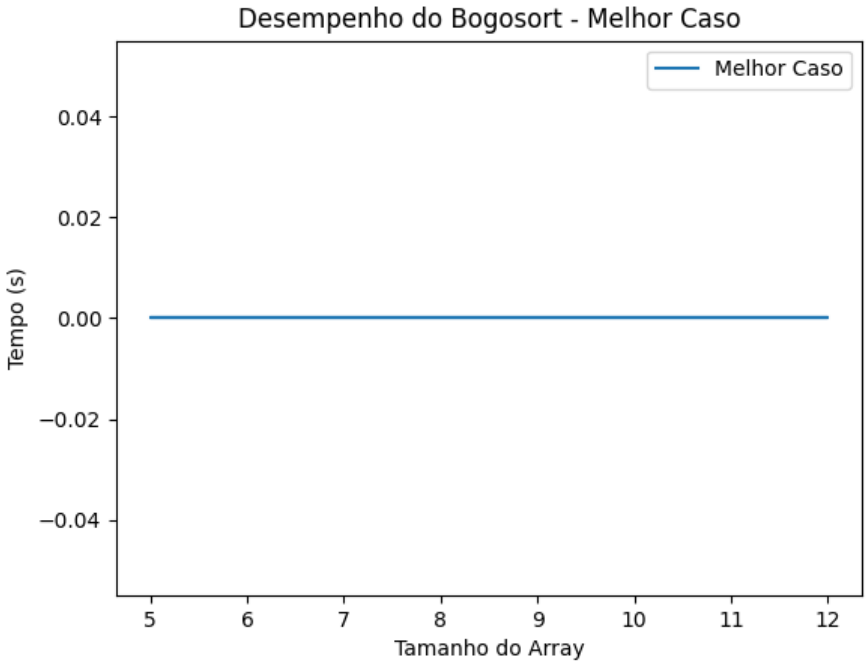
Tamanho do vetor 12: 95.206000 ssegundos

```
Array original (Pior Caso - Tamanho 12):
11 10 9 8 7 6 5 4 3 2 1 0
Array ordenado (Pior Caso - Tamanho 12):
0 1 2 3 4 5 6 7 8 9 10 11
Tempo para ordenar: 95.206000 segundos
```

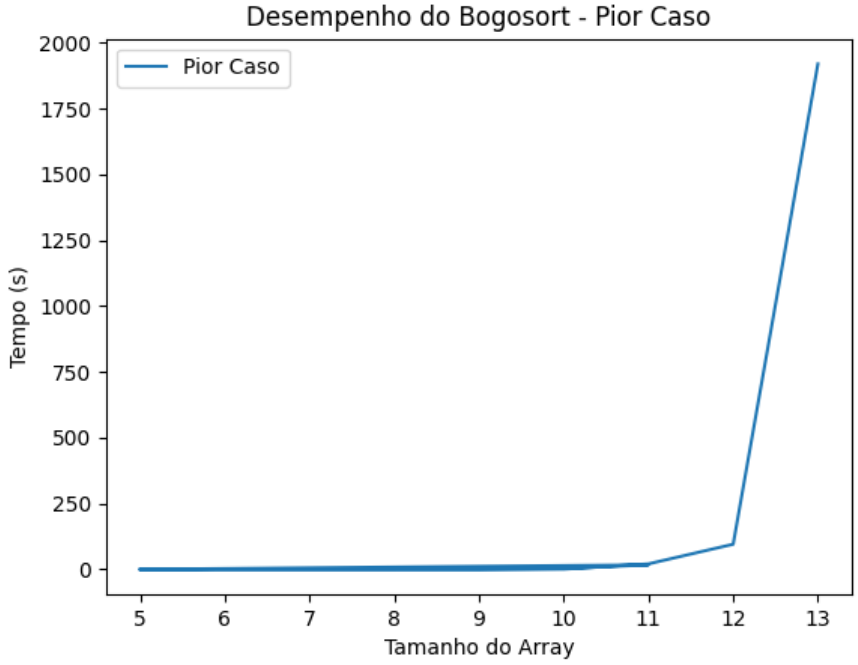
Tamanho do vetor 13: 32 minutos

```
Array original (Pior Caso - Tamanho 13):
12 11 10 9 8 7 6 5 4 3 2 1 0
Array ordenado (Pior Caso - Tamanho 13):
0 1 2 3 4 5 6 7 8 9 10 11 12
Tempo para ordenar: 1919.664000 segundos
```

Melhor caso:



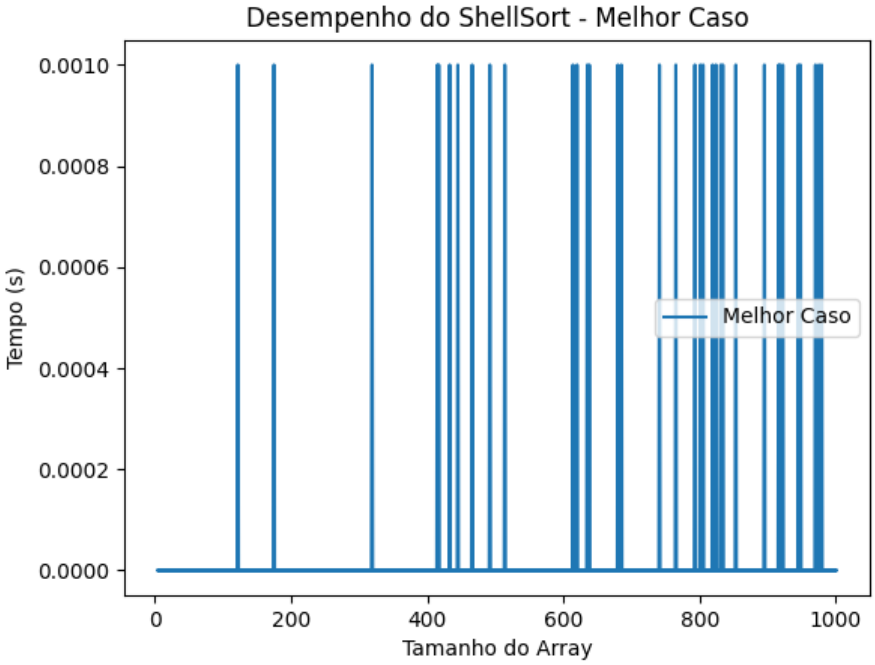
Pior Caso:



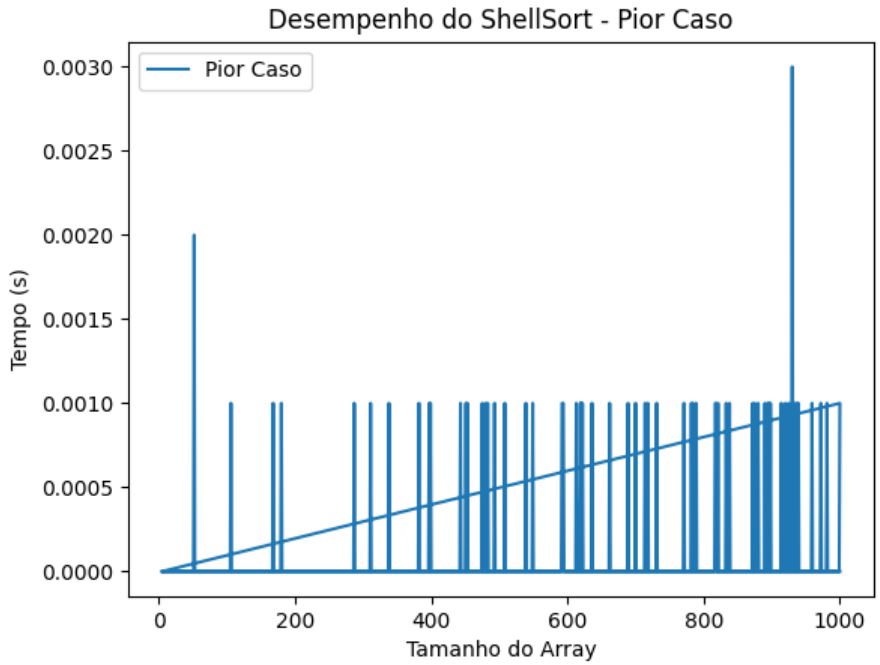
4.2 ShellSort

Os testes no ShellSort variaram bastante de acordo com os dados, por mais que tiveram variações no gráfico as variações foram pequenas levando em consideração o tempo.

Melhor Caso:



Pior Caso:



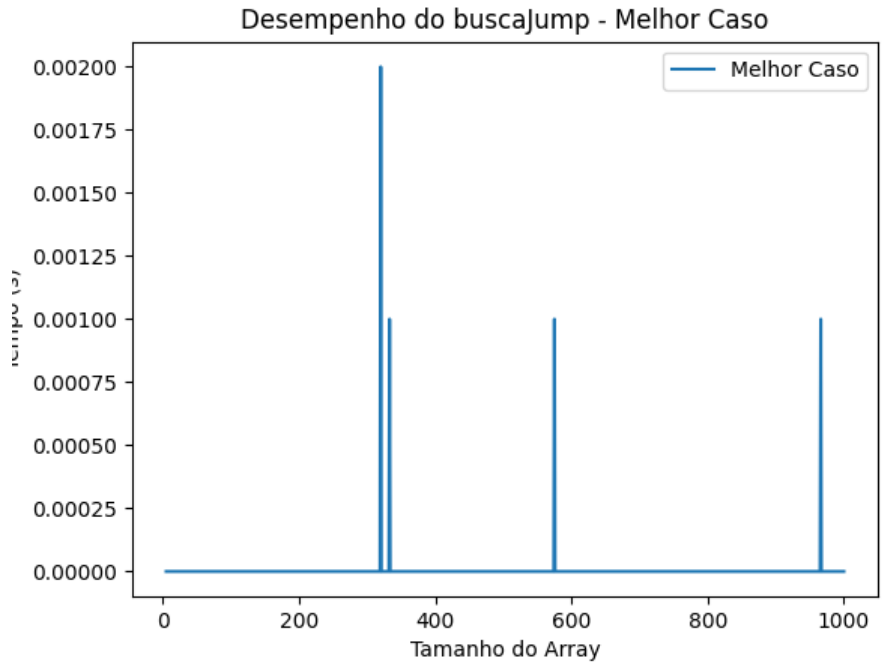
Análise: O ShellSort mostrou um aumento relativamente suave no tempo de execução à medida que o tamanho da entrada aumentava. Isso confirma a maior eficiência do ShellSort em comparação com o BogoSort, especialmente para tamanhos de entrada maiores.

4.3 Busca por Jump

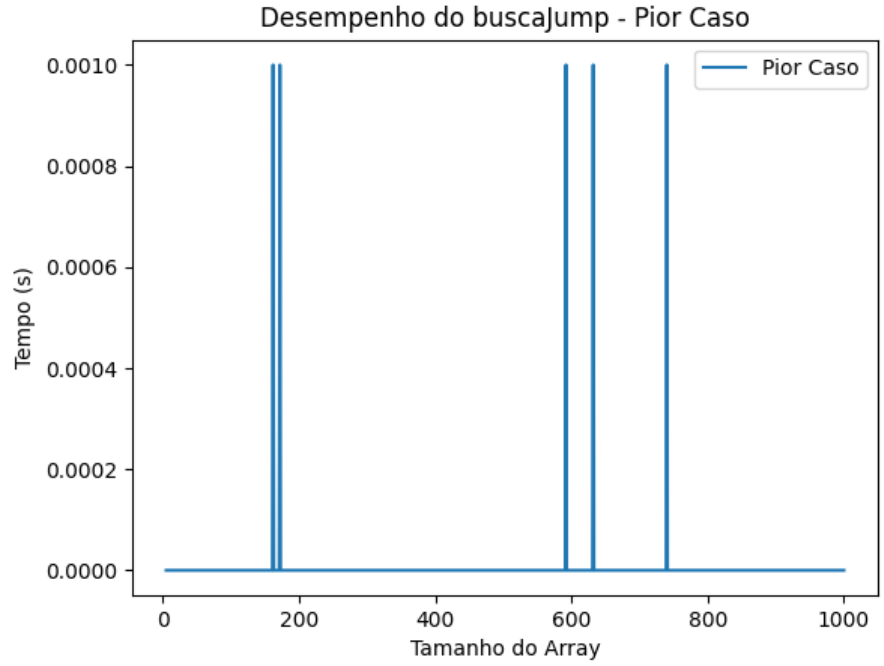
Os testes no JumpSearch variaram bastante de acordo com os dados, por mais que tiveram variações no gráfico as variações foram pequenas levando em consideração o tempo.

Os testes foram realizados com tamanhos de lista variados.

Melhor caso:



Pior Caso:

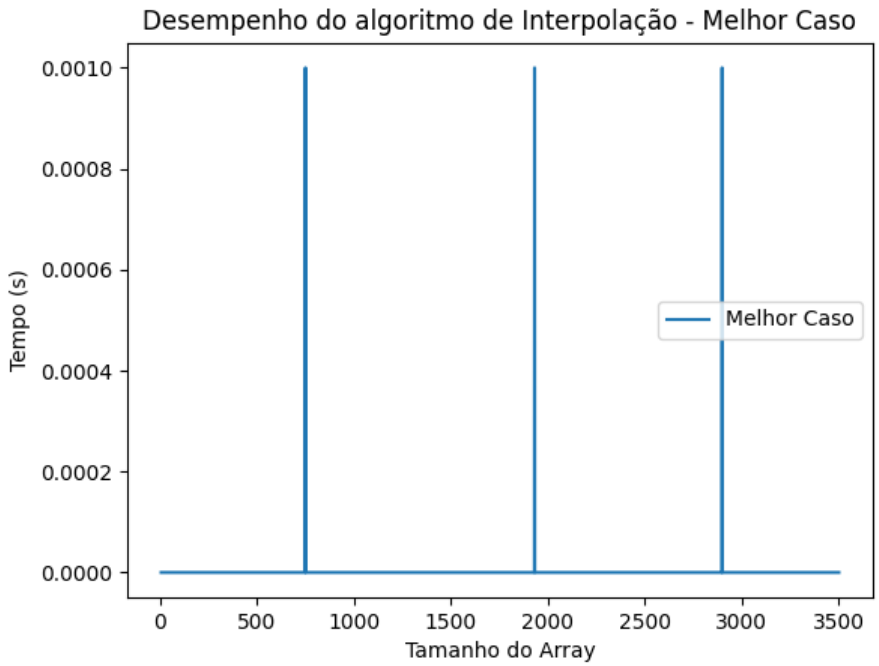


Análise: O algoritmo de Busca por Jump apresenta um aumento linear no tempo de execução à medida que o tamanho do array aumenta, o que é esperado dada sua complexidade O(√n). A busca por Jump é eficiente para arrays maiores, mas seu desempenho é influenciado pelo tamanho dos passos, que é a raiz quadrada do tamanho do array.

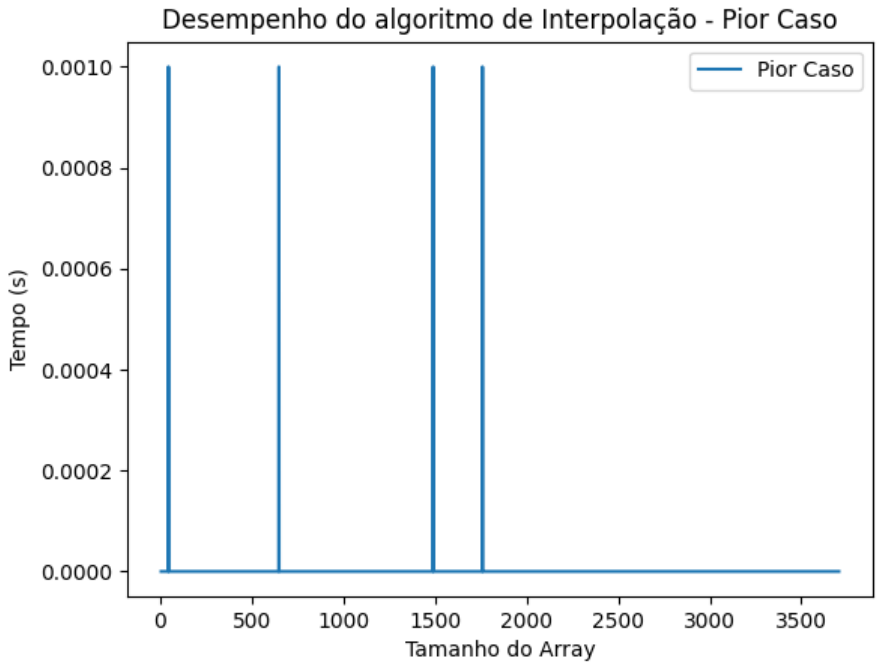
4.4 Busca por Interpolação

Os testes foram realizados com tamanhos de lista variados.

Melhor Caso:



Pior Caso:



Análise: A Busca por Interpolação mostrou um tempo de execução extremamente baixo e consistente em todos os tamanhos de array testados. Este comportamento exemplifica a eficiência da Busca por Interpolação em arrays ordenados com distribuição uniforme dos elementos. A complexidade teórica de O(log log n) é observada na prática, especialmente em grandes arrays.

5 CONCLUSÕES

BogoSort: O Bogo Sort é um algoritmo de ordenação notoriamente ineficiente, com complexidade média exponencial, tornando-o impraticável para conjuntos de dados de tamanho significativo. Seu objetivo é simplesmente embaralhar aleatoriamente os elementos até que a lista esteja ordenada. Devido à sua natureza extremamente ineficiente, o Bogo Sort é mais frequentemente usado como uma curiosidade ou desafio em vez de uma solução prática. Sua utilização é limitada a contextos humorísticos ou desafios de programação, e não é recomendado para aplicações do mundo real.

ShellSort: O Shell Sort é um algoritmo de ordenação que visa melhorar a eficiência do algoritmo de ordenação por inserção, especialmente para conjuntos de dados de grande tamanho. Ao dividir o conjunto de dados em intervalos menores e aplicar o algoritmo de ordenação por inserção a esses subconjuntos, o Shell Sort consegue reduzir a quantidade de trocas necessárias para ordenar a lista. Sua complexidade média é consideravelmente melhor do que a do Bogo Sort, tornando-o

uma escolha viável para conjuntos de dados moderadamente grandes em comparação com outros algoritmos mais eficientes. O Shell Sort é utilizado quando a eficiência do algoritmo de ordenação por inserção é desejada em conjuntos de dados de tamanho moderado, e a simplicidade de implementação é um fator importante.

Jump Search: O Jump Search é um algoritmo de busca eficiente para conjuntos de dados ordenados. Ao pular blocos fixos, o Jump Search reduz o número de comparações necessárias para encontrar um elemento em comparação com a busca linear. Com uma complexidade média de $O(\sqrt{n})$, o Jump Search é uma escolha eficiente para grandes conjuntos de dados, especialmente quando uma busca binária não é prática devido à impossibilidade de acessar elementos arbitrários, como em listas vinculadas. Sua utilização é adequada para situações em que é possível estimar onde o elemento desejado pode estar e quando a eficiência é crucial.

Busca por Interpolação: A Busca por Interpolação é um algoritmo de busca eficiente para conjuntos de dados uniformemente distribuídos. Ao ajustar a estimativa interpolada da posição do elemento desejado com base na distribuição dos valores, a Busca por Interpolação pode oferecer uma complexidade média mais eficiente ($O(\log \log n)$) em comparação com outros algoritmos de busca. Sua utilização é apropriada quando o conjunto de dados possui uma distribuição uniforme, proporcionando benefícios significativos nesse cenário. No entanto, em conjuntos de dados não uniformemente distribuídos, a Busca por Interpolação pode não apresentar vantagens consideráveis em relação a outros algoritmos de busca.

5 REFERÊNCIAS BIBLIOGRAFICAS

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms. MIT Press.

Knuth, D. E. (1997). The Art of Computer Programming, Volume 3: Sorting and Searching. Addison-Wesley.