Relatório Projeto Entregável 05 - Tiro no escuro

João Pedro Neves e Juan Pablo Tomba

24 de outubro de 2025

1 Introdução

Neste relatório, explicaremos o processo de criação de uma versão de código que visa resolver o seguinte problema: ordenar N valores ordenados de forma aleatória, ordenada e inversamente ordenada.

Serão discutidos aspectos como tempo de execução, descrição de como cada algoritmo funciona e discussão sobre suas complexidades.

2 Metodologia

O primeiro passo do projeto foi criar um repositório público no GitHub onde ambos os membros do grupo poderiam subir os códigos. Dessa forma, poderíamos desenvolver simultaneamente os diferentes códigos e ao final ter uma forma prática de juntá-los e/ou acessá-los.

https://github.com/joaopneves1570/Lab-ICC2/tree/main/aula6

Em seguida, criados os arquivos para o projeto, utilizamos o ambiente de desenvolvimento do Visual Studio Code para escrever e compilar os mesmos.

3 Código

Após isso, elaboramos o seguinte código:

```c
#include "util.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <math.h>

void gerar_reverso(int n, int* sequencia) {
    for (int i = 0; i < n; i++)
        sequencia[i] = n - i;
}

void gerar_aleatoria(int n, int* sequencia) {
    int seed = 12345;
    for (int i = 0; i < n; i++)
        sequencia[i] = get_random(&seed, n);
}

void gerar_ordenada(int n, int* sequencia) {
    for (int i = 0; i < n; i++)
        sequencia[i] = i + 1;
}

static inline void swap(int* a, int* b) {
    int t = *a;
    *a = *b;
    *b = t;
}

/* -------------------- SHELL SORT -------------------- */
void shellSort(int s[], int n) {
```

```
    for (int gap = n / 2; gap > 0; gap /= 2) {
        for (int i = gap; i < n; i++) {
            int temp = s[i];
            int j;
            for (j = i; j >= gap && s[j - gap] > temp; j -= gap)
                s[j] = s[j - gap];
            s[j] = temp;
        }
    }
}

/* -------------------- QUICK SORT -------------------- */
void quickSort(int v[], int inf, int sup) {
    if (inf >= sup) return;

    int i = inf, j = sup;
    int pivo = v[(inf + sup) / 2];

    while (i <= j) {
        while (v[i] < pivo) i++;
        while (v[j] > pivo) j--;
        if (i <= j) {
            swap(&v[i], &v[j]);
            i++;
            j--;
        }
    }

    if (inf < j) quickSort(v, inf, j);
    if (i < sup) quickSort(v, i, sup);
}

/* -------------------- HEAP SORT -------------------- */
void rearranjar_heap(int v[], int i, int tamanho) {
    register int temp = v[i];
    register int filho;

    while ((filho = 2 * i + 1) < tamanho) {
        if (filho + 1 < tamanho && v[filho + 1] > v[filho])
            filho++;

        if (v[filho] > temp) {
            v[i] = v[filho];
            i = filho;
        } else {
            break;
        }
    }

    v[i] = temp;
}

void heapsort(int v[], int n) {
    for (int i = n / 2 - 1; i >= 0; i--)
        rearranjar_heap(v, i, n);
    for (int i = n - 1; i > 0; i--) {
        swap(&v[0], &v[i]);
        rearranjar_heap(v, 0, i);
    }
}

/* -------------------- INTRO SORT -------------------- */

int depthLimit(int n) {
    return (int)(2.0 * log(n));
}
```

```c

void introsortUtil(int v[], int inicio, int fim, int depthLimit) {
    int tamanho = fim - inicio + 1;

    if (depthLimit == 0) {
        heapsort(v + inicio, tamanho);
        return;
    }

    int i = inicio, j = fim;
    int pivo = v[(inicio + fim) / 2];

    while (i <= j) {
        while (v[i] < pivo) i++;
        while (v[j] > pivo) j--;
        if (i <= j) {
            swap(&v[i], &v[j]);
            i++;
            j--;
        }
    }

    if (inicio < j) introsortUtil(v, inicio, j, depthLimit - 1);
    if (i < fim) introsortUtil(v, i, fim, depthLimit - 1);
}

void introsort(int v[], int n) {
    int limit = depthLimit(n);
    introsortUtil(v, 0, n - 1, limit);
}



/* -------------------- MAIN -------------------- */
int main() {
    int n;
    scanf("%d", &n);

    char comando[16];
    scanf("%15s", comando);

    int algoritmo;
    scanf("%d", &algoritmo);

    int* sequencia = (int*)malloc(sizeof(int) * n);
    if (!sequencia) return 1;

    if (strcmp(comando, "reverse") == 0)
        gerar_reverso(n, sequencia);
    else if (strcmp(comando, "random") == 0)
        gerar_aleatoria(n, sequencia);
    else if (strcmp(comando, "sorted") == 0)
        gerar_ordenada(n, sequencia);


    switch (algoritmo) {
        case 1:
            quickSort(sequencia, 0, n - 1);
            break;
        case 2:
            shellSort(sequencia, n);
            break;
        case 3:
            introsort(sequencia, n);
            break;
        }
```

```
164
165
166     init_crc32();
167     uint32_t saida = crc32(0, sequencia, n * sizeof(int));
168     printf("%08X", saida);
169
170     free(sequencia);
171     return 0;
172 }
```

4 Funcionamento de cada algoritmo

Nesta seção, explicaremos como cada algoritmo utilizado funciona:

4.1 Shell Sort

Funcionamento: O Shell Sort é uma melhoria do Insertion Sort. Ele inicialmente compara elementos a uma distância **H**, reduzindo esse valor a cada iteração até que o algoritmo se torne um Insertion Sort tradicional. Isso permite que elementos distantes sejam reposicionados rapidamente, reduzindo o número total de deslocamentos.

Complexidades:

- Melhor caso: $O(n \log n)$
- Caso médio:$O(n^{3/2})$ dependendo da escolha do **H**
- Pior caso: $O(n^2)$
- **Obs:** o desempenho do Shell Sort varia bastante conforme a escolha do **H** utilizada.

4.2 Quick Sort

Funcionamento: Utiliza divisão e conquista. Escolhe-se um pivô, e o vetor é particionado em duas partes: elementos menores ficam à esquerda e maiores à direita. O processo é realizado recursivamente em ambas as partições. Apesar de muito eficiente em média, o Quick Sort **não é estável** e pode apresentar piora de desempenho dependendo do pivô.

Complexidades:

- Melhor caso: $O(n \log n)$
- Caso médio: $O(n \log n)$
- Pior caso: $O(n^2)$ — acontece quando o pivô divide o vetor de forma altamente desigual

4.3 IntroSort

Funcionamento: É um algoritmo híbrido que combina Quick Sort e Heap Sort. Seu funcionamento inicia com o Quick Sort pela eficiência em média, mas monitora a profundidade da recursão. Se ela ultrapassar um limite definido ($2 \log n$), indicando risco de cair no pior caso do Quick Sort, o algoritmo muda automaticamente para o Heap Sort, garantindo bom desempenho.

Complexidades:

- Melhor caso: $O(n \log n)$
- Caso médio: $O(n \log n)$
- Pior caso: $O(n \log n)$ — pois evita o pior caso do Quick Sort

5 Resultados

5.1 Tempo de Execução

Para medir os tempos de execução, fizemos o upload do código no RunCodes e de lá retiramos o tempo de execução fornecido pela própria plataforma.

- **Caso 1**: 1.8831 s (reverso - quickSort)
- **Caso 2**: 3.7173 s (reverso - shellSort)
- **Caso 3**: 1.5388 s (reverso - introSort)

Obs: em todos os casos, são 60000000 elementos, inversamente ordenados

6 Discussão dos Resultados

Nesta seção, discutiremos por que apenas os três algoritmos escolhidos — Shell Sort, Quick Sort e IntroSort — passaram nos testes de tempo do RunCodes.

Por que esses três algoritmos passaram e os outros não?

Algoritmos como Bubble Sort, Insertion Sort e Selection Sort possuem complexidade quadrática $O(n^2)$ no pior caso. Para uma entrada de tamanho tão grande, o tempo de execução desses métodos seria impraticável, resultando facilmente em horas ou dias de processamento.

No entanto, os algoritmos utilizados possuem complexidades que, na prática, se aproximam de $O(n \log n)$:

- Shell Sort — embora possa ter pior caso $O(n^2)$, apresentou desempenho aceitável devido à redução de deslocamentos usando diferentes valores de H.
- Quick Sort — extremamente eficiente em média, apesar do risco de pior caso quadrático.
- IntroSort — combina Quick Sort e Heap Sort, garantindo $O(n \log n)$ em qualquer caso.

Assim, estes três algoritmos conseguem ordenar grandes volumes de dados dentro do tempo limite dos testes automatizados.

Qual foi o mais rápido e por quê?

Com base nos resultados, todos os testes foram realizados com vetores **inversamente ordenados**, o que interfere diretamente no comportamento dos algoritmos.

O IntroSort foi o mais rápido por possuir uma adaptação: ele detecta quando o Quick Sort pode entrar em seu pior caso e troca de abordagem utilizando Heap Sort, mantendo desempenho elevado.

O Quick Sort teve boa performance, mas sofreu com algumas partições desfavoráveis devido à escolha do pivô como elemento central — o que não garante uma divisão balanceada em vetores reversos.

O Shell Sort, apesar de ser mais eficiente que algoritmos quadráticos, ainda depende fortemente da escolha da sequência de H, o que o torna menos eficiente em entradas grandes.

De forma geral, os resultados reforçam que algoritmos adaptativos como o IntroSort, costumam obter os melhores tempos em entradas muito grandes.

7 Conclusão

Os resultados obtidos demonstram que, ao lidar com entradas muito grandes e desfavoráveis — como vetores inversamente ordenados — algoritmos com comportamento próximo a $O(n \log n)$ tornam-se essenciais para execução dentro de tempos aceitáveis. Nesse cenário, o IntroSort se destacou como a solução mais eficiente, exatamente por evitar o pior caso do Quick Sort através da alternância para o Heap Sort quando necessário.

O Quick Sort também apresentou bom desempenho, embora mais sensível à escolha do pivô, enquanto o Shell Sort, apesar de eficaz para tamanhos moderados, mostrou menor eficiência no processamento de milhões de elementos.

Assim, a análise mostra a importância de escolher algoritmos adequados de acordo com o tamanho e características da entrada.