Atividade 5 - Tiro no Escuro (tiro certeiro)

Davi Gabriel Domingues - 15447497

13 de Outubro de 2025

1 Explicações do cenário

O algoritmo contido em "tiroNoEscuro.c" lida com uma situação peculiar: processos de ordenação otimizados o suficiente para lidarem com contextos de entradas de dados consideravelmente grandes. Para se resolver do melhor jeito possível, foram realizadas diversas otimizações, a partir de algoritmos de ordenação secundários, assim como pelo uso de partições e de deslocamentos específicos, a fim de se amortizar o custo de desempenho computacional.

Listing 1: Função `swap` — troca de valores

```c
// Troca simples de dois inteiros por referencia
static void swap_int(int *a, int *b) {
    int t = *a;
    *a = *b;
    *b = t;
}
```

Listing 2: Método de ordenação secundário — Insertion Sort

```c
// Insertion Sort in-place em [lo, hi]
// Bom desempenho para subarranjos pequenos (usado como cutoff no Quicksort)
static void insertion_sort(int *a, int lo, int hi) {
    // Complexidade O(m^2) no pior caso (m = hi - lo + 1); porem muito eficiente para
        subarranjos pequenos.
    // Util como "cutoff" para reduzir overhead de recursao/merges em algoritmos hibridos.
    for (int i = lo + 1; i <= hi; ++i) {
        int key = a[i], j = i - 1;

        // Desloca elementos maiores que key uma posicao a direita
        // Invariante: a[lo..j] ja esta ordenado e todos > key serao movidos
        while (j >= lo && a[j] > key) {
            a[j + 1] = a[j];
            --j;
        }

        a[j + 1] = key;
    }
}
```

Listing 3: Método de troca secundário — mediana de 3

```c
// Escolha de pivo por mediana de tres: a[lo], a[mid], a[hi]
// Reduz a chance de piores casos em entradas quase ordenadas
static int median_of_three(int *a, int lo, int hi) {
    // Seleciona a mediana entre a[lo], a[mid], a[hi] e a move para a[lo] como pivo.
    // Ajuda a evitar piores casos em entradas ja (quase) ordenadas.
```

```
    int mid = lo + ((hi - lo) >> 1);
    if (a[mid] < a[lo])
        swap_int(&a[mid], &a[lo]);

    if (a[hi] < a[mid])
        swap_int(&a[hi], &a[mid]);

    if (a[mid] < a[lo])
        swap_int(&a[mid], &a[lo]);

    // Move o pivo (mediana) para a[lo]
    swap_int(&a[lo], &a[mid]);
    return a[lo];
}
```

Listing 4: Método de troca terciário – partição de Hoare

```
// Particionamento de Hoare em [lo, hi]
// Retorna indice j tal que [lo, j] <= pivo e [j+1, hi] >= pivo
static int partition_hoare(int *a, int lo, int hi) {
    // Particionamento de Hoare:
    // - Mantem i avancando ate a[i] >= pivo e j recuando ate a[j] <= pivo; troca quando
        i < j.
    // - Retorna j tal que [lo..j] <= pivo e [j+1..hi] >= pivo (intervalos podem se
        sobrepor em valores iguais).
    int pivot = median_of_three(a, lo, hi), i = lo - 1, j = hi + 1;

    for (;;) {
        // Avanca i ate encontrar elemento >= pivo
        do {
            ++i;
        } while (a[i] < pivot);

        // Regride j ate encontrar elemento <= pivo
        do {
            --j;
        } while (a[j] > pivot);

        if (i >= j)
            return j; // regiao particionada

        swap_int(&a[i], &a[j]);
    }
}
```

Listing 5: Função de checagem para vetor ordenado

```
// Helpers para detectar ordenacao e reverter rapidamente casos degenerados
static int is_sorted_asc(const int *a, int n) {
    // Checagem linear O(n) usada como fast-path para evitar trabalho desnecessario
    for (int i = 1; i < n; ++i) {
        if (a[i - 1] > a[i]) {
            return 0;
        }
    }
```

```
    return 1;
}
```

Listing 6: Função de checagem para vetor inversamente ordenado

```
static int is_sorted_desc(const int *a, int n) {
    // Varredura linear O(n) para detectar ordem estritamente decrescente
    for (int i = 1; i < n; ++i) {
        if (a[i - 1] < a[i])
            return 0;
    }

    return 1;
}
```

Sendo assim, podemos lidar com os desempenhos de cada algoritmo de ordenação utilizado de forma separada, dado às suas construções diferenciadas, sendo, então notório destacar os seguintes âmbitos:

1.1 Quick Sort

O seu funcionamento básico/essencial segue o mesmo padrão descrito no relatório passado (entrega04), porém foram implementadas algumas mudanças para otimizar o processo de ordenação do método em si:

Listing 7: Implementação do algoritmo Quick Sort (otimizado)

```
// Quicksort com:
// - cutoff para insertion sort quando o subarray e pequeno
// - particionamento de Hoare
// - recursao na menor particao (tail recursion elimination)
static void quicksort_impl(int *a, int lo, int hi) {
    // Estrategias:
    // - Cutoff para insertion sort quando subarray e pequeno (reduz overhead e melhora
        localidade)
    // - Particionamento de Hoare (menos swaps em media)
    // - Recursao sempre na menor metade (eliminacao de recursao de cauda), limitando
        profundidade para O(log n)
    while (lo < hi) {
        if (hi - lo + 1 <= 16) { // cutoff para pequenos subarrays (valor empirico)
            insertion_sort(a, lo, hi);
            break;
        }

        int p = partition_hoare(a, lo, hi);

        // Ordena recursivamente a menor metade para limitar profundidade
        if (p - lo < hi - (p + 1)) {
            quicksort_impl(a, lo, p);
            lo = p + 1; // itera na metade menor
        }

        else {
            quicksort_impl(a, p + 1, hi);
```

```
            hi = p; // itera na metade maior
        }
    }
}
```

1.2 Merge Sort

O seu funcionamento básico/essencial segue o mesmo padrão descrito no relatório passado (entrega04), porém foram implementadas algumas mudanças para otimizar o processo de ordenação do método em si:

Listing 8: Implementação do algoritmo Quick Sort (otimizado)

```
// Mergesort iterativo (bottom-up) hibridizado com insertion sort para runs pequenas
// - Pre-ordena runs de tamanho RUN com insertion sort
// - Faz merge em "ping-pong" entre array e tmp, copiando ao final so se necessario
void algoritmo_id2_mergesort(int *array, int n) {
    // Versao iterativa (bottom-up) com runs pequenas otimizadas via insertion sort
        (melhora localidade e cache).
    if (n <= 1)
        return;

    // Fast-paths: ja ordenado crescente ou totalmente decrescente
    if (is_sorted_asc(array, n))
        return;

    if (is_sorted_desc(array, n)) {
        reverse_array(array, n);
        return;
    }

    size_t N = (size_t)n;

    // Pre-ordena runs pequenas com insertion sort
    const size_t RUN = 32; // tamanho de run escolhido empiricamente; ajustavel conforme
        arquitetura e dados

    for (size_t i = 0; i < N; i += RUN) {
        size_t lo = i;
        size_t hi = (i + RUN - 1 < N) ? (i + RUN - 1) : (N - 1);
        insertion_sort(array, (int)lo, (int)hi);
    }

    // Se ja coube tudo em uma run, ja esta ordenado
    if (N <= RUN) return;

    int *tmp = (int *)malloc(N * sizeof(int));

    int *src = array, *dst = tmp;
    size_t width = RUN;
    while (width < N) {
        // Pre-checagem: se todas as fronteiras de runs ja estao em ordem, termina
        // Evita passes de merge desnecessarios quando quase tudo ja esta mesclado
        int already_sorted = 1;
```

```
    for (size_t i = 0; i + width < N; i += (width << 1)) {
        size_t mid = i + width;
        if (src[mid - 1] > src[mid]) { already_sorted = 0; break; }
    }

    if (already_sorted) break;

    int any_merged = 0;
    for (size_t i = 0; i < N; i += (width << 1)) {
        size_t left = i, mid = i + width, right = i + (width << 1);
        if (mid > N)
            mid = N;

        if (right > N)
            right = N;

        // Se bloco direito vazio ou ja em ordem, so copia
        if (mid >= right || src[mid - 1] <= src[mid]) {
            memcpy(&dst[left], &src[left], (right - left) * sizeof(int));
            continue;
        }

        // Merge estavel src -> dst
        // Invariante: [left..mid) e [mid..right) estao ordenados em src
        size_t p = left, q = mid, k = left;
        while (p < mid && q < right) {
            if (src[p] <= src[q])
                dst[k++] = src[p++];

            else
            dst[k++] = src[q++];
        }

        // Copia o restante (um dos lados sempre esgota primeiro)
        while (p < mid)
            dst[k++] = src[p++];

        while (q < right)
            dst[k++] = src[q++];

        any_merged = 1;
    }

    if (!any_merged) break; // nada a mesclar neste passe

    // Proximo passe: troca src/dst (ping-pong). Evita copias desnecessarias a cada
        nivel.
    int *swap = src; src = dst; dst = swap;

    if (width > (N >> 1)) break;  // evita overflow ao dobrar
    width <<= 1;
}

// Se o resultado final ficou em tmp (src != array), copia de volta uma vez
if (src != array)
    memcpy(array, src, N * sizeof(int));
```

```
    free(tmp);
}
```

Obs: Nesse cenário, foi aplicada a versão iterativa do Merge Sort, diferentemente da presente na entrega04, a qual era de caráter recursivo (padrão mais conhecido). O mergesort do arquivo é uma versão iterativa e prática que combina insertion sort em runs pequenas, detecção de casos fáceis e um esquema ping-pong com memcpy/checagens para minimizar trabalho e cópias, conferindo maior estabilidade e robustez com um bom desempenho, sem sacrificar a complexidade O(n log n).

1.3 Heap Sort

O Heapsort do arquivo é implementado na função `algoritmo_id3_heapsort` e segue o esquema clássico em duas fases, construindo um *max-heap* e depois extrai repetidamente o máximo termo, o colocando no fim do array. Para a situação apresentada, houve algumas otimizações práticas para reduzir as trocas e o trabalho em casos pequenos ou quase ordenados. Logo no início da função há *fast-paths* que evitam trabalho desnecessário: o código verifica se o array já está ordenado em ordem crescente e retorna imediatamente; se estiver totalmente decrescente, chama `reverse_array` e retorna. Essas checagens (funções `is_sorted_asc`, `is_sorted_desc` e `reverse_array`) estão definidas no mesmo ficheiro e usadas antes de qualquer construção de heap.

Em seguida há um *cutoff* para arrays muito pequenos: se `n <= 32` a função chama insertion_sort(array, 0, n-1) e sai, isto é um híbrido comum porque o insertion sort costuma ser mais rápido em blocos pequenos por causa do baixo overhead e melhor localidade de memória; a implementação de `insertion_sort` e seu comentário explicativo também aparecem no arquivo/ficheiro. A construção da heap é feita pelo método de Floyd: o loop `for (int i = (n » 1) - 1; i >= 0; -i) sift_down(array, n, i);` aplica `sift_down` a todos os nós internos de forma decrescente, transformando o array em um *max-heap* em tempo linear $O(n)$. Essa linha e o comentário a associando ao método de Floyd estão diretamente dentro de `algoritmo_id3_heapsort`.

A rotina/função `sift_down` implementa o «down-heap» otimizado: ela *cacheia* o valor a ser afundado em `v = a[i]`, percorre enquanto há filhos (usando algo como `half = n » 1`), escolhe o maior e o move para cima, sem fazer swaps repetidos, e só no final coloca `v` na posição correta. Esse padrão reduz atribuições e trocas desnecessárias, além de melhorar a localidade do código (a definição de `sift_down` e o comentário explicativo estão no código em si).

Na fase de extração, o laço `for (int end = n - 1; end > 0; -end)` retira o máximo e restaura a heap. Em vez de fazer um swap `root`↔`último` e depois `sift_down(0)`, o código usa a técnica que economiza um swap: guarda `max = array[0]` e `v = array[end]`, chama `sift_down_with_val(array, end, 0, v)` (variante que começa a afundar `v` a partir da raiz), e só então escreve `array[end] = max`, evitando, assim, uma troca inicial e reduzindo o total de movimentações. Dentro do mesmo laço, há outra otimização híbrida: se `end < 32` o código interrompe o ciclo e finaliza a ordenação do prefixo restante com `insertion_sort(array, 0, end)`. Isso aproveita que, no final do heapsort, quando o heap diminui bastante, é mais eficiente terminar com insertion sort do que continuar com muitos `sift_down`s. A verificação `if (end < 32) { insertion_sort(...); break; }` aparece explicitamente em `algoritmo_id3_heapsort`.

Por fim, o código contém comentários e uma pequena documentação no topo que explicam o trade-off desta implementação: Heapsort aqui é in-place e garante $O(n \log n)$ no pior caso, mas não é estável; as otimizações (*fast-paths*, cutoff para insertion sort, `sift_down` que cacheia o valor e `sift_down_with_val`) existem para reduzir swaps, melhorar a localidade e para acelerar os casos reais.

Listing 9: Implementação do algoritmo Quick Sort (otimizado)

```c
// Restaura a propriedade de heap maximo a partir do indice i em um heap de tamanho n
static void sift_down(int *a, int n, int i) {
    // "Sift-down" classico de heap maximo. Usa um cache do valor (v) para reduzir swaps.
    int v = a[i], half = n >> 1; // primeiros 'half' indices sao nos internos

    while (i < half) {
        int l = (i << 1) + 1, r = l + 1;

        // escolhe o maior filho
        int j = (r < n && a[r] > a[l]) ? r : l;

        if (a[j] <= v)
            break; // posicao correta encontrada

        a[i] = a[j];
        i = j;
    }

    a[i] = v;
}

// Variante de sift-down que insere um valor 'v' comecando na raiz (i)
// em um heap de tamanho n, evitando o swap no laco principal do sort.
static void sift_down_with_val(int *a, int n, int i, int v) {
    // Variante que evita a troca inicial entre raiz e ultimo elemento.
    // 'v' e o valor que sera afundado a partir da posicao 'i'.
    int half = n >> 1;

    while (i < half) {
        int l = (i << 1) + 1, r = l + 1, j = (r < n && a[r] > a[l]) ? r : l;

        if (a[j] <= v)
            break;

        a[i] = a[j];
        i = j;
    }

    a[i] = v;
}

// Heapsort hibridizado com insertion sort para tamanhos pequenos e cauda pequena
void algoritmo_id3_heapsort(int *array, int n) {
    if (n <= 1) return;

    // Fast-paths: ja ordenado crescente ou totalmente decrescente
    if (is_sorted_asc(array, n))
        return;

    if (is_sorted_desc(array, n)) {
        reverse_array(array, n);
        return;
    }

    // Cutoff para tamanhos pequenos: insertion sort e tipicamente melhor em cache
```

```
    if (n <= 32) {
        insertion_sort(array, 0, n - 1);
        return;
    }

    // Construcao do heap (Floyd): O(n)
    for (int i = (n >> 1) - 1; i >= 0; --i)
        sift_down(array, n, i);

    // Extracao do maximo + restauracao do heap: O(n log n)
    // Cutoff: termina com insertion sort quando a cauda for pequena
    for (int end = n - 1; end > 0; --end) {
        if (end < 32) { // ordena o prefixo restante de forma eficiente
            insertion_sort(array, 0, end);
            break;
        }

        int max = array[0], v = array[end];

        sift_down_with_val(array, end, 0, v);
        array[end] = max;
    }
}
```

2 Comparação do tempo de execução dos métodos de ordenação

Uma boa maneira de avaliar as vantagens e desvantagens de cada algoritmo é comparar o tempo de execução deles, sendo que utilizaremos 3 categorizações de ordenação, para o contexto apresentado:

- Grupo 1: Conjunto já ordenado.
- Grupo 2: Conjunto em ordem aleatória.
- Grupo 3: Conjunto ordenado inversamente.

Esses tipos de dados, assim como os seus contextos de aplicação como entrada para cada um dos algoritmos, foram aplicados e suas quantidades foram devidamente variadas, sendo que os resultados obtidos foram colocados na tabela abaixo:

Grupo	Quick Sort	Merge Sort	Heap Sort
1	2.781	0.806	0.829
2	1.043	1.385	2.393
3	1.882	0.804	0.837

Table 1: Tempos por algoritmo e arquivo de teste, medidos em segundos

Os resultados mostram comportamentos distintos conforme o tipo da entrada. Mergesort e Heapsort apresentam tempos muito baixos nas entradas já ordenadas e ordenadas inversamente

porque suas implementações incluem *fast-paths* que detectam vetores já ordenados (`is_sorted_asc`) ou estritamente decrescentes (`is_sorted_desc`) e retornam (ou chamam `reverse_array`) sem executar a rotina completa de ordenação. Dessa forma, nesses casos, o custo efetivo fica próximo de uma varredura linear O(n).

Quicksort, por outro lado, não aplica essas verificações iniciais no wrapper; portanto, mesmo em vetores já ordenados ele realiza particionamentos e iterações até o cutoff, o que provoca sobrecarga adicional e explica o tempo maior observado no caso "já ordenado". Em entradas aleatórias, Quicksort foi o mais rápido graças às otimizações de mediana-de-três, ao particionamento de Hoare, ao cutoff para `insertion_sort` e à eliminação de recursão de cauda — combinações que reduzem as constantes e favorecem o desempenho médio.

Heapsort, apesar de fornecer garantia teórica de $O(n \log n)$ no pior caso, obteve o pior tempo no caso aleatório. Isso é esperado: a implementação faz muitos acessos não-sequenciais (pais/filhos no array), o que prejudica a localidade de cache e aumenta os custos práticos. Já nos casos ordenados/inversos o Heapsort se beneficia dos fast-paths presentes no código e, portanto, aparece competitivo.

Dessa forma, podemos concluir que:

- **Mergesort** é o mais robusto quando há entrada já ordenada ou inversa (graças às detecções rápidas);
- **Quicksort** é o melhor na média (aleatório) por ter menores constantes e boas heurísticas;
- **Heapsort** garante pior caso, porém no experimento prático sofreu com maior custo por acesso de memória, o que o tornou o algoritmo mais lento para o caso de entradas aleatórias.

2.1 Por que esses três algoritmos passaram nos testes e por que os outros não?

Os três algoritmos testados — **Quicksort**, **Mergesort** e **Heapsort** — passaram nos testes por causa de três fatores combinados:

- **Tratamento de casos de borda e de validação:** as implementações incluem verificações explícitas para entradas triviais (como $n \leq 1$), testes de ordenação prévia (`is_sorted_asc`, `is_sorted_desc`) e, ao final, validação do vetor ordenado. Isso elimina erros comuns e evita executar rotinas completas desnecessariamente, o que aumenta a chance de passar os testes de corretude e de desempenho.
- **Heurísticas maduras e implementação testada:** Quicksort contém mediana-de-três, particionamento robusto e recursão na menor metade, Mergesort e Heapsort têm versões iterativas/híbridas com `insertion_sort` para blocos pequenos. Essas técnicas reduzem casos patológicos e constantes práticas, tornando as versões testadas mais confiáveis.
- **Higiene de código e de controle de intervalos:** as funções usam índices bem definidos, evitam off-by-one e tratam subarrays vazios/pequenos explicitamente — erros típicos que costumam causar falhas em testes automatizados.

Em contraste, implementações que não passaram, como o Radix Sort e o Counting Sort, tendem a falhar por causas recorrentes e bem conhecidas:

- **Bugs de limites ou de partição:** off-by-one em índices, limites de laço incorretos ou condição de partição mal formulada (p.ex. falha ao separar corretamente subarrays vazios),

considerando a complexidade envolvida no tamanho absoluto de dados de entrada, aumentam as chances de se produzir arrays incorretamente ordenados.

- **Falta de controle de recursão:** a ausência de parada correta para $n \leq 1$ ou recursão não protegida podem causar comportamento indefinido ou estouro de pilha.