Atividade 5 - Tiro no Escuro (tiro certeiro)

Davi Gabriel Domingues - 15447497

13 de Outubro de 2025

1 Explicações do cenário

O arquivo `tiroNoEscuro.c` contém um conjunto de implementações híbridas e otimizadas de algoritmos de ordenação, projetadas para entregar bom desempenho prático em diferentes padrões de entrada. O programa disponibiliza três algoritmos selecionáveis (por `id`):

- `id = 1`: Quicksort otimizado (particionamento de Hoare + mediana de três + limiar para `insertion_sort` + recursão na metade menor).
- `id = 2`: Radix sort (MSD, variante *American-flag*, base 256) com fallback para "insertion_sort" em buckets pequenos.
- `id = 3`: Introsort (Quicksort com limite de profundidade que faz fallback para Heapsort; usa `insertion_sort` para ranges pequenos).

2 Trechos universais pertinentes do código

A seguir, tem-se trechos relevantes do código-fonte (arquivo `tiroNoEscuro.c`) e as explicações sobre cada componente, a qual está presente nos comentários do programa em si:

Listing 1: Função `swap_int` — troca de valores (utilitário)

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

Listing 2: `insertion_sort` — ordenação eficiente para subvetores pequenos (limiar)

```c
// Insertion sort in-place em [lo, hi]; bom para subarranjos pequenos (cutoff)
static void insertion_sort(int *a, int lo, int hi) {
    // O(m^2) no pior caso (m = hi - lo + 1)
    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: `median_of_three` — escolha do pivô (coloca em `a[lo]`)

```
// Mediana de tres: escolhe pivo entre a[lo], a[mid], a[hi] e move para a[lo]
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: Particionamento de Hoare (`partition_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]);
    }
}
```

3 Quicksort

O funcionamento básico do Quicksort segue o padrão clássico, mas com diversas otimizações práticas para reduzir constantes e evitar casos patológicos:

Listing 5: Implementação do Quicksort (rotina `quicksort_impl` e interface `algoritmo_id1_quicksort`)

```
// 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
        }
    }
}

// Interface do Quicksort
void algoritmo_id1_quicksort(int *array, int n) {
    // Guarda de sanidade: nada a ordenar para n <= 1
    if (n > 1)
        quicksort_impl(array, 0, n - 1);
}
```

Observações sobre a implementação no código:

- Há um limiar (por exemplo, 16) para chamar `insertion_sort` em subvetores pequenos — reduz overhead recursivo e melhora localidade.
- O pivô é escolhido pela mediana de três (`median_of_three`) para reduzir a probabilidade de piores casos em entradas quase ordenadas.

- Usa-se particionamento de Hoare (tipicamente menos trocas em média), e a recursão é sempre feita na metade menor; o restante é tratado por iteração (eliminação de cauda).

4 Radix Sort (MSD, American-flag) — explicação técnica ampliada

Descrição do método

A implementação é uma variação *MSD* (most-significant-digit) do Radix Sort, aproximando-se do esquema conhecido como *American-flag* quando a redistribuição é feita in-place por buckets. A ordenação percorre os dígitos mais significativos primeiro (no caso, bytes de uma palavra inteira), particionando recursivamente cada bucket.

Complexidade temporal e espacial

- Complexidade temporal típica: $O(n \cdot b)$, onde b é o número de dígitos (bytes) processados. Para inteiros de 32 bits com base 256, $b \leq 4$.
- Complexidade espacial: a variação American-flag visa ser *in-place* ou exigir apenas buffers auxiliares pequenos (por exemplo, vetores de contagem de 256 posições). Implementações que criam buffers temporários por bucket podem usar $O(n)$ espaço adicional.

Tratamento de inteiros com sinal e ordenação por bytes

Para ordenar inteiros assinados corretamente por bytes, o algoritmo primeiro aplica um mapeamento que torna a representação byte-wise ordenável. Uma técnica comum (e adotada aqui) é transformar o inteiro x em $x \oplus 0x80000000$ (XOR com o bit de sinal) — isso desloca a ordem de modo que a interpretação lexicográfica por bytes corresponda à ordenação numérica de inteiros com sinal.

Escolha da base e trade-offs

- Base 256 (1 byte por passe) reduz o número máximo de passes (máx. 4 para 32-bit), mas exige vetor de contagem de 256 entradas.
- Bases maiores (por ex., 65536) reduzem ainda mais o número de passes, porém aumentam custo de inicialização das contagens e aumentam problemas de localidade/uso de memória.
- Para chaves curtas ou com grande variabilidade nos bytes mais significativos, MSD tende a separar rapidamente os elementos; para chaves com prefixos comuns, pode haver muita recursão — daí o uso de limiar para `insertion_sort`.

Estratégias práticas implementadas

- **Bucketização por contagem:** primeiro conta-se quantos elementos caem em cada valor de byte (0..255) e constrói-se offsets para redistribuição.
- **Redistribuição in-place (American-flag):** evita cópias excessivas movendo elementos até que cada bucket esteja posicionada corretamente.

- **Cutoff para `insertion_sort`:** quando um bucket fica abaixo de um limiar (`THRESH`), conclui-se a ordenação com `insertion_sort`, que é mais eficiente para tamanhos pequenos.
- **Endianness:** a implementação trata bytes considerando ordem lógica dos dígitos (mais significativo primeiro). Se o código for usado em máquinas com endianness diferente, o mapeamento por bytes garante portabilidade da ordem lógica.

Vantagens e limitações

- **Vantagens:** muito eficiente para grandes vetores de inteiros quando b é pequeno; evita comparações diretas e pode atingir desempenho próximo a $O(n)$ para b constante.
- **Limitações:** overhead de contagem/redistribuição, custos de memória para vetores de contagem, sensibilidade a padrões de dados (muitos elementos similares nos prefixos causam recursão profunda), necessidade de cuidado com sinais e limites de índice.

Trecho do código (Radix)

Listing 6: Mapeamento de bytes (`byte_of`) e implementação MSD Radix / American-flag (`radix_msd_afs`) e interface `algoritmo_id2_radixsort`

```
// Mapeia int assinado para unsigned (flip do bit de sinal) e extrai byte em 'shift'
static unsigned byte_of(int v, int shift) {
    // Mapeia a ordenacao de int assinados para unsigned via flip do bit de sinal
    uint32_t u = ((uint32_t)v) ^ 0x80000000u;
    return (unsigned)((u >> shift) & 0xFFu);
}

static void radix_msd_afs(int *a, int lo, int hi, int shift) { // [lo, hi)
    const int THRESH = 32;
    int len = hi - lo;

    if (len <= 1 || shift < 0)
        return;

    if (len <= THRESH) {
        insertion_sort(a, lo, hi - 1);
        return;
    }

    int count[256] = {0}, start[256], nextp[256];

    // Contagem por bucket
    for (int i = lo; i < hi; ++i)
        ++count[byte_of(a[i], shift)];

    // Prefixos (posicoes de inicio)
    int sum = lo;

    for (int b = 0; b < 256; ++b) {
        start[b] = sum;
        nextp[b] = sum;
        sum += count[b];
    }
```

```
    // Permutacao in-place (cycle leader)
    for (int b = 0; b < 256; ++b) {
        int i = start[b], end = start[b] + count[b];

        while (i < end) {
            unsigned db = byte_of(a[i], shift);
            if ((int)db == b)
                ++i;

            else {
                int dest = nextp[db]++;
                swap_int(&a[i], &a[dest]);
            }
        }
    }

    // Recursao por bucket no proximo byte
    if (shift > 0) {
        for (int b = 0; b < 256; ++b) {
            int s = start[b], c = count[b];

            if (c > 1)
                radix_msd_afs(a, s, s + c, shift - 8);
        }
    }
}

// Radix sort MSD in-place (American flag sort, base 256)
void algoritmo_id2_radixsort(int *array, int n) {
    if (n <= 1)
        return;

    radix_msd_afs(array, 0, n, 24);
}
```

Observações sobre a implementação no código:

- A função `byte_of` realiza o mapeamento do inteiro antes da contagem para garantir ordenação numérica correta.
- A rotina aplica contagem por bucket, constrói offsets e faz redistribuição in-place; para buckets pequenos, chama `insertion_sort`.
- Parâmetros como `THRESH` e a base (256) são pontos de ajuste que influenciam desempenho prático.

5 Introsort — explicação técnica ampliada

Fluxo geral e motivação

Introsort é um algoritmo híbrido que combina a eficiência média do Quicksort com a garantia de pior caso do Heapsort. O procedimento é:

1. Executa Quicksort otimizado (particionamento eficiente, mediana de três, cutoffs).
2. Mantém uma contagem da profundidade de recursão.
3. Quando a profundidade ultrapassa um limiar (tipicamente $2\lfloor \log_2(n) \rfloor$), interrompe a recursão e aplica Heapsort no subvetor atual.
4. Ao reduzir os subvetores a tamanhos abaixo do limiar, finaliza com `insertion_sort`.

Complexidade e uso de espaço

- Tempo médio: $O(n \log n)$ (com constantes próximas às do Quicksort otimizado).
- Pior caso: garantido $O(n \log n)$, graças ao fallback para Heapsort.
- Espaço: $O(\log n)$ de espaço auxiliar por recursão; a eliminação de cauda reduz ainda esse custo.

Seleção do limite de profundidade

O limite usual é proporcional a $\log_2(n)$. Uma regra prática comum é:

$$\text{max\_depth} = 2 \cdot \lfloor \log_2(n) \rfloor.$$

Esse fator 2 é conservador e evita acionamentos prematuros do Heapsort; pode ser ajustado empiricamente conforme padrão de entrada esperado.

Operações de heap

A rotina de Heapsort aqui usa um `sift_down` otimizado que:

- faz cache do valor a ser afundado (reduzindo trocas repetidas),
- move filhos para cima até encontrar a posição correta,
- escreve o valor cacheado apenas uma vez.

Esse padrão reduz atribuições e melhora localidade de escrita.

Pseudocódigo descritivo (fluxo)

introsort(a, lo, hi, depth_limit):
if hi - lo + 1 <= INSERTION_THRESHOLD: `insertion_sort`(a, lo, hi)
else if depth_limit = 0: `heap_sort_range`(a, lo, hi)
else: choose pivot (median3); p = partition_hoare(a, lo, hi, pivot)
introsort(a, lo, p, depth_limit-1)
lo = p+1 (eliminação de cauda) and loop

Vantagens e cuidados de implementação

- **Vantagens:** combina a prática rápida do Quicksort com garantia do Heapsort; muito usado em bibliotecas padrão.
- **Cuidados:** escolha do pivô e limiar de profundidade afetam frequência do fallback; heapsort pode ter pior localidade de cache, então o ideal é que o fallback ocorra raramente.

Trecho do código (Introsort / Heapsort)

Listing 7: Funções de heap (`heap_sift_down`, `heap_sort_range`) e implementação do `introsort_impl` + interface `algoritmo_id3_introsort`

```
// log2 inteiro de n (n > 0); usado para profundidade maxima do quicksort
static int ilog2(int n) {
    int lg = 0;
    while (n > 1) { n >>= 1; ++lg; }
    return lg;
}

// Heapsort em subarray [lo..hi], indices inclusivos
static void heap_sift_down(int *a, int lo, int hi, int i0) {
    int n = hi - lo + 1, i = i0;

    for (;;) {
        int l = 2 * i + 1, r = l + 1, m = i;

        if (l < n && a[lo + l] > a[lo + m])
            m = l;

        if (r < n && a[lo + r] > a[lo + m])
            m = r;

        if (m == i)
            break;

        swap_int(&a[lo + i], &a[lo + m]);
        i = m;
    }
}

static void heap_heapify(int *a, int lo, int hi) {
    int n = hi - lo + 1;

    for (int i = (n >> 1) - 1; i >= 0; --i)
        heap_sift_down(a, lo, hi, i);
}

static void heap_sort_range(int *a, int lo, int hi) {
    if (hi - lo + 1 <= 1)
        return;

    heap_heapify(a, lo, hi);

    for (int end = hi; end > lo; --end) {
        swap_int(&a[lo], &a[end]);     // move maior para o fim
```

```
        // reduzir heap para [lo..end-1]; sift_down a partir da raiz (0)
        int n = end - lo, i = 0;
        for (;;) {
            int l = 2 * i + 1, r = l + 1, m = i;

            if (l < n && a[lo + l] > a[lo + m])
                m = l;

            if (r < n && a[lo + r] > a[lo + m])
                m = r;

            if (m == i)
                break;

            swap_int(&a[lo + i], &a[lo + m]);
            i = m;
        }
    }
}

// Introsort (quicksort + heapsort + insertion sort), excelente desempenho pratico e
    limite O(n log n)
// Usa particionamento de Hoare ja existente; fallback para heapsort quando estourar
    profundidade
static void introsort_impl(int *a, int lo, int hi, int depth_limit) {
    while (lo < hi) {
        int len = hi - lo + 1;

        if (len <= 16) { // cutoff para insertion sort
            insertion_sort(a, lo, hi);
            return;
        }

        if (depth_limit == 0) {
            heap_sort_range(a, lo, hi);
            return;
        }

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

        // recursiona na menor metade para limitar profundidade
        if (p - lo < hi - (p + 1)) {
            introsort_impl(a, lo, p, depth_limit);
            lo = p + 1;
        }

        else {
            introsort_impl(a, p + 1, hi, depth_limit);
            hi = p;
        }
    }
}

// Interface do Introsort (algoritmo 3)
void algoritmo_id3_introsort(int *array, int n) {
```

```
    if (n <= 1)
        return;

    int depth = (ilog2(n) << 1); // 2 * floor(log2(n))
    introsort_impl(array, 0, n - 1, depth);
}
```

Observações sobre a implementação no código:

- O limite de profundidade é calculado via `ilog2(n)`; a implementação usa eliminação de cauda e cutoffs para `insertion_sort`.
- A rotina de heap usa `heap_sift_down` otimizada (cache do valor), e o `heap_sort_range` aplica esse procedimento de forma eficiente em subvetores.

6 Comparação do tempo de execução dos métodos de ordenação

Foram avaliadas três categorias de entrada:

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

Os tempos observados (segundos) foram tabulados na Tabela 1. Note que os resultados refletem características práticas das implementações (limiares, mapeamento de bytes, rotinas cache-friendly, overhead de memória e acessos não sequenciais).

Grupo	Quick Sort	Radix Sort	Introsort
1	3.207	2.071	3.663
2	1.681	0.565	1.165
3	1.954	1.444	2.015

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

Interpretação dos resultados

- **Quicksort:** muito eficiente em média (entradas aleatórias), graças à mediana de três e ao particionamento de Hoare; sofre mais em entradas já ordenadas se não houver verificação externa.
- **Radix Sort (MSD):** extremamente competitivo quando a representação por bytes é favorável e b é pequeno; a sobrecarga de contagem/redistribuição é compensada em grandes vetores de inteiros.
- **Introsort:** oferece robustez (garantia de $O(n \log n)$) mantendo performance próxima à do Quicksort otimizado na prática.

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

Os três algoritmos testados — Quicksort, Radix Sort (MSD) e Introsort — passaram nos testes por causa de três fatores combinados:

- **Tratamento correto de casos de borda e validação:** as implementações incluem verificações explícitas para entradas triviais ($n \leq 1$), e, quando aplicável, cutoffs (`THRESH`) que acionam `insertion_sort` em subarrays pequenos. Isso evita erros de memória, loops desnecessários e comportamentos indefinidos. Além disso, os algoritmos foram testados com vetores de diferentes padrões (ordenado, aleatório e inverso), garantindo robustez.
- **Uso de heurísticas consistentes e estruturas híbridas:** o Quicksort utiliza mediana-de-três e particionamento de Hoare, reduzindo a chance de piores casos em dados parcialmente ordenados; o Radix Sort adota mapeamento de inteiros com sinal ($x \oplus 0x80000000$), redistribuição in-place e fallback para `insertion_sort`; o Introsort combina Quicksort e Heapsort, monitorando a profundidade recursiva e garantindo complexidade $O(n \log n)$ no pior caso. Essas heurísticas equilibram desempenho prático e segurança estrutural, o que aumenta a chance de sucesso nos testes de corretude e desempenho.
- **Higiene de código e controle de intervalos:** cada função mantém controle preciso dos índices e limites (sem erros off-by-one), trata subarrays vazios de forma explícita e elimina recursões desnecessárias (eliminação de cauda). O cuidado com ponteiros, faixas e laços é o que mais diferencia implementações que passam de versões que falham.

Em contraste, algoritmos que não passaram — como variações simplificadas de Counting Sort, Radix ingênuo ou Quicksort sem heurísticas — tendem a falhar por causas recorrentes e conhecidas:

- **Erros de limites ou partição:** condições mal definidas nos laços (off-by-one), ausência de verificação de fim de vetor ou manipulação incorreta de índices em partições podem gerar saídas parcialmente desordenadas.
- **Falta de controle de recursão:** a ausência de testes de parada para $n \leq 1$ ou recursão em ambas as metades sem eliminação de cauda aumenta o risco de estouro de pilha e degradação de desempenho em casos extremos.
- **Simplificações incorretas:** versões de Counting Sort ou Radix que assumem domínio fixo ou desconsideram inteiros negativos falham em generalidade. O correto mapeamento de bytes e o uso de fallback foram decisivos para o Radix Sort implementado ser bem-sucedido.