Algoritmos de Ordenação

Merge Sort

Aulas 5 e 6

Estrutura de Dados - 3º Semestre ADS

Revisão das Aulas Anteriores

Algoritmos de Ordenação Estudados

Antes de avançarmos para o Merge Sort, vamos revisar brevemente os algoritmos que já estudamos:

Algoritmo	Estratégia	Melhor Caso	Caso Médio	Pior Caso	Espaço	Estável
Bubble Sort	Troca	O(n)	O(n^2)	O(n^2)	O(1)	✓
Selection Sort	Seleção	O(n^2)	O(n^2)	O(n^2)	O(1)	✗
Insertion Sort	Inserção	O(n)	O(n^2)	O(n^2)	O(1)	✓

Bubble Sort

Compara pares adjacentes e os troca se estiverem na ordem errada, fazendo os maiores "flutuarem" para o final.

Características: Simples, muitas trocas, ineficiente para grandes conjuntos.

Selection Sort

Encontra o menor elemento e o coloca na posição correta, expandindo a parte ordenada gradualmente.

Características: Poucas trocas, muitas comparações, desempenho previsível.

Insertion Sort

Constrói a sequência ordenada um elemento por vez, inserindo cada elemento na posição correta.

Características: Adaptativo, eficiente para pequenos conjuntos ou quase ordenados.

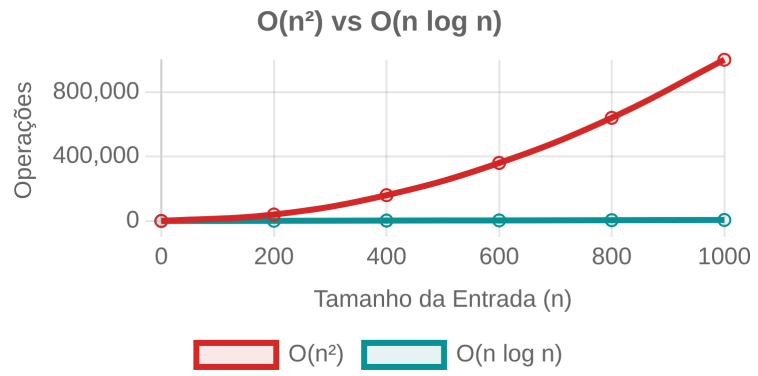
Limitações dos Algoritmos O(n^2)

Os algoritmos que estudamos até agora têm complexidade quadrática O(n^2) no caso médio e pior caso, o que os torna ineficientes para grandes conjuntos de dados.

Para um array com 1 milhão de elementos:

- Bubble/Selection/Insertion Sort: ~10^{12} operações
- Merge Sort (que veremos hoje): ~2×10^7 operações

Precisamos de algoritmos mais eficientes!



Paradigma: Divisão e Conquista

Um Novo Paradigma de Algoritmos

Antes de entender o Merge Sort, precisamos conhecer o paradigma de **Divisão e Conquista** (Divide and Conquer):

O que é Divisão e Conquista?

Uma estratégia algorítmica que resolve problemas complexos dividindo-os em subproblemas menores e mais simples, resolvendo cada subproblema independentemente e depois combinando as soluções para obter a solução do problema original.

1 Dividir

Dividir o problema original em subproblemas menores, geralmente de tamanho igual ou similar.

Exemplo: Dividir um array em duas metades.

2 Conquistar

Resolver os subproblemas recursivamente. Se os subproblemas forem pequenos o suficiente, resolvê-los diretamente.

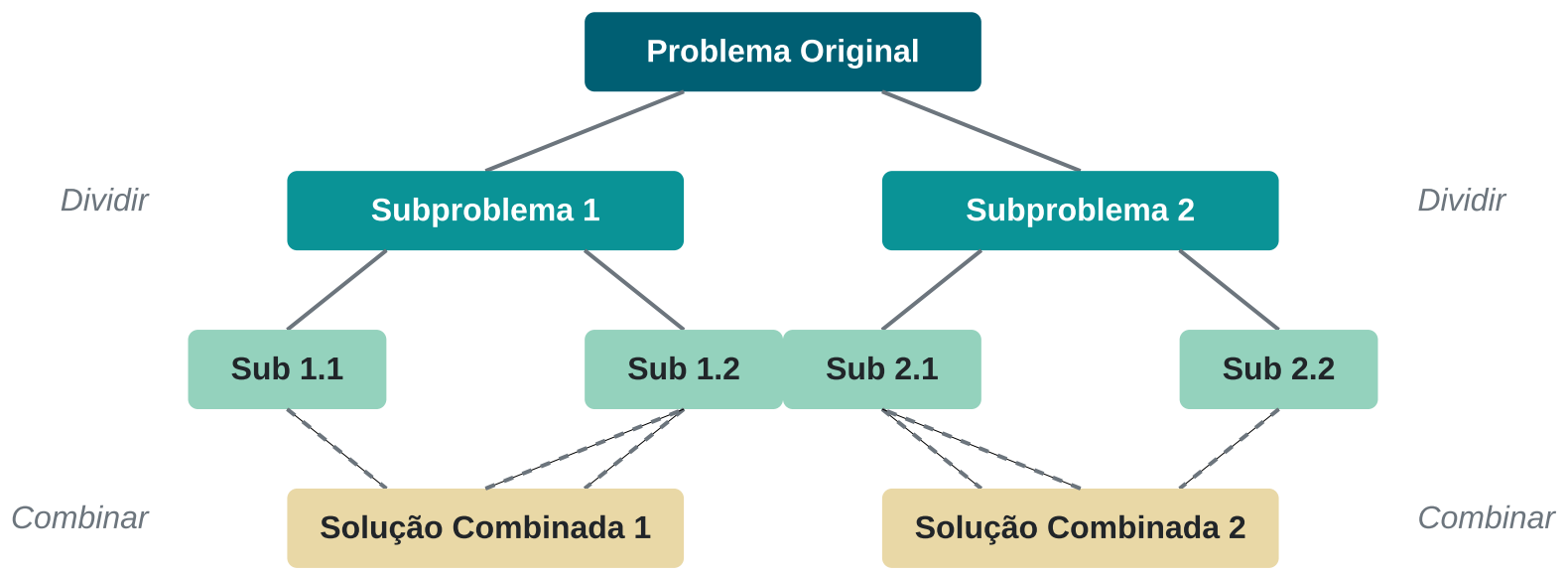
Exemplo: Ordenar cada metade do array.

3 Combinar

Combinar as soluções dos subproblemas para obter a solução do problema original.

Exemplo: Mesclar as duas metades ordenadas.

Visualização do Paradigma



Algoritmos Baseados em Divisão e Conquista

Merge Sort

Divide o array, ordena as partes e depois as mescla.

Quick Sort

Particiona o array em torno de um pivô e ordena as partições.

Busca Binária

Divide o espaço de busca pela metade a cada iteração.

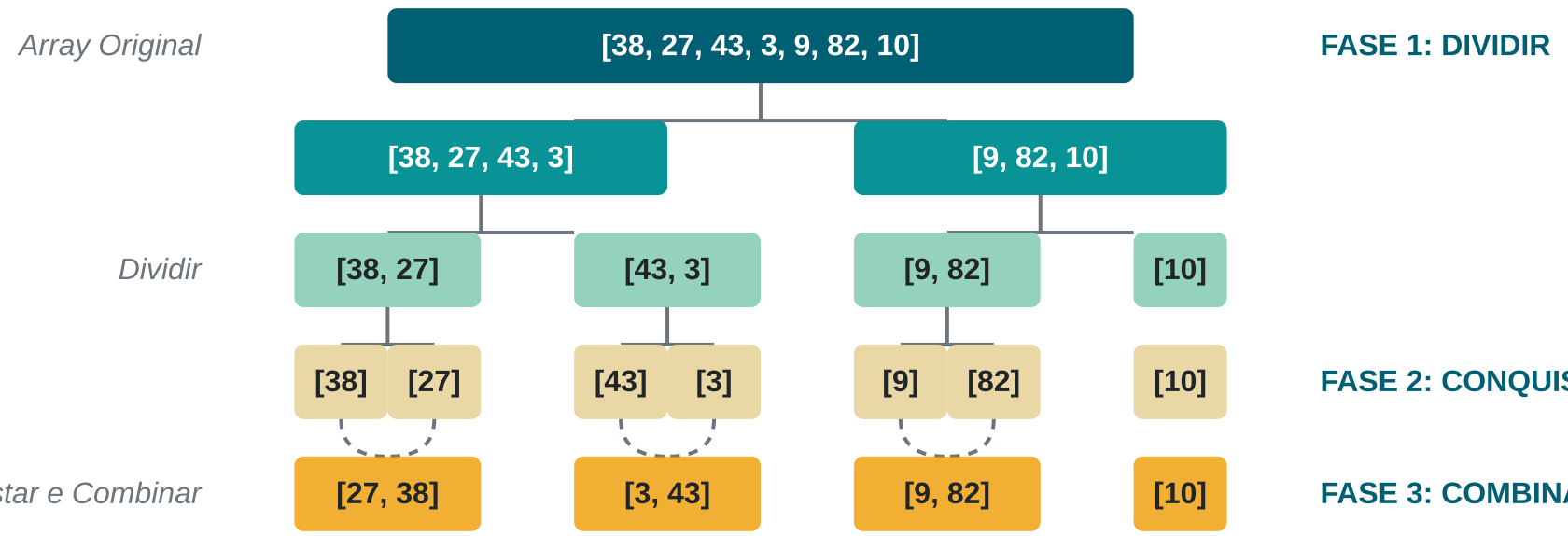
Merge Sort: Conceito

O Algoritmo Merge Sort

O Merge Sort é um algoritmo de ordenação eficiente baseado no paradigma de **Divisão e Conquista**:

> ### Ideia Fundamental
>
> Dividir o array em metades menores, ordenar cada metade recursivamente e, em seguida, mesclar (merge) as metades ordenadas para produzir o array final ordenado.

Visão Geral do Processo



Características Principais

- **Estabilidade:** Mantém a ordem relativa de elementos iguais
- **Complexidade:** O(n log n) em todos os casos (melhor, médio e pior)
- **Memória:** Requer espaço auxiliar O(n) - não é in-place
- **Recursividade:** Implementação típica usa recursão
- **Paralelizável:** Pode ser facilmente paralelizado

Vantagens e Desvantagens

Vantagens

- Desempenho previsível
- Eficiente para grandes conjuntos
- Estável
- Bom para dados externos

Desvantagens

- Uso de memória extra
- Overhead para pequenos arrays
- Não é in-place

O Merge Sort é um dos primeiros algoritmos de ordenação eficientes que estudaremos, com complexidade O(n log n), significativamente melhor que os algoritmos $O(n^2)$ vistos anteriormente.

Merge Sort: Algoritmo

Algoritmo Passo a Passo

Vamos detalhar o funcionamento do Merge Sort com um exemplo completo:

Pseudocódigo

```
função mergeSort(arr[], inicio, fim):
    se inicio < fim:
        // Encontra o ponto médio do array
        meio = (inicio + fim) / 2

        // Ordena a primeira metade
        mergeSort(arr, inicio, meio)

        // Ordena a segunda metade
        mergeSort(arr, meio + 1, fim)

        // Mescla as duas metades ordenadas
        merge(arr, inicio, meio, fim)

função merge(arr[], inicio, meio, fim):
    // Cria arrays temporários
    n1 = meio - inicio + 1
    n2 = fim - meio

    L[] = novo array de tamanho n1
    R[] = novo array de tamanho n2

    // Copia dados para arrays temporários
    para i = 0 até n1-1:
        L[i] = arr[inicio + i]
    para j = 0 até n2-1:
        R[j] = arr[meio + 1 + j]

    // Mescla os arrays temporários
    i = 0, j = 0, k = inicio
    enquanto i < n1 e j < n2:
        se L[i] <= R[j]:
            arr[k] = L[i]
            i++
        senão:
            arr[k] = R[j]
            j++
        k++

    // Copia elementos restantes de L[]
    enquanto i < n1:
        arr[k] = L[i]
        i++
        k++

    // Copia elementos restantes de R[]
    enquanto j < n2:
        arr[k] = R[j]
        j++
        k++
```

Exemplo de Execução

Array: [38, 27, 43, 3, 9, 82, 10]

Fase 1: Divisão Recursiva

Nível 1: Array completo

38	27	43	3	9	82	10

Nível 2: Dividido em duas metades

38	27	43	3

9	82	10

Nível 3: Subdivisões menores

38	27

43	3

9	82

10

Fase 2: Mesclagem Recursiva

Nível 3: Mesclagem de pares

27	38

3	43

9	82

10

Nível 2: Mesclagem de quartetos

3	27	38	43

9	10	82

Nível 1: Mesclagem final

3	9	10	27	38	43	82

Explicação

Função mergeSort()

Função recursiva que divide o array ao meio, ordena cada metade e depois mescla as partes ordenadas.

Função merge()

Função que mescla duas partes ordenadas do array em uma única sequência ordenada.

Arrays Temporários

Usados para armazenar as duas metades durante o processo de mesclagem.

Processo de Mesclagem

Compara elementos dos dois arrays temporários e os coloca na posição correta no array original.

Merge Sort: Processo de Mescla

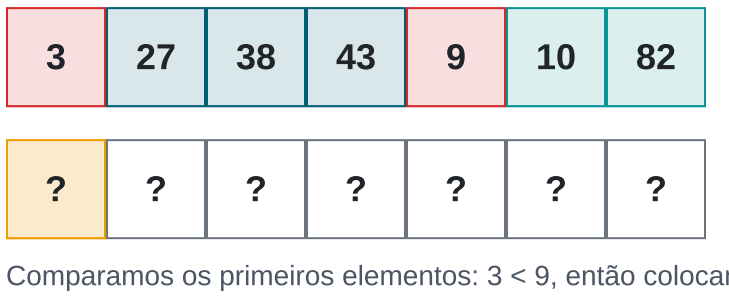
Detalhamento do Processo de Mescla

O coração do Merge Sort está na função **merge()**, que combina duas sublistas ordenadas em uma única lista ordenada:

Exemplo Passo a Passo

Vamos mesclar duas sublistas ordenadas: [3, 27, 38, 43] e [9, 10, 82]

Passo 1: Inicialização

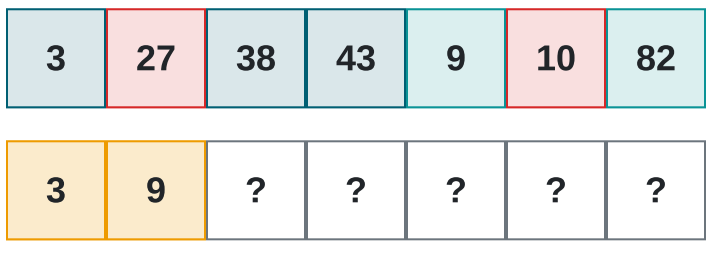


Comparamos os primeiros elementos: 3 < 9, então colocamos 3 no array resultado.

Passo 2: Próxima comparação

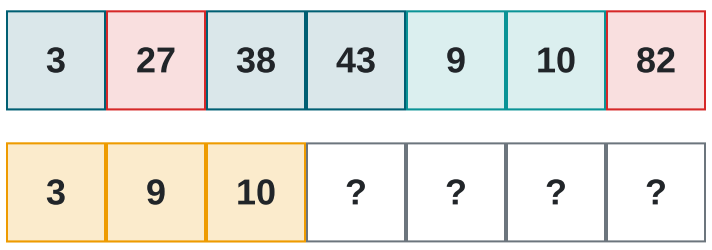
Comparamos os próximos elementos: 27 > 9, então colocamos 9 no array resultado.

Passo 3: Continuando



Comparamos os próximos elementos: 27 > 10, então colocamos 10 no array resultado.

Passo 4: Continuando



Comparamos os próximos elementos: 27 < 82, então colocamos 27 no array resultado.

Resultado Final

Continuamos o processo até que todos os elementos sejam colocados no array resultado.

Detalhes do Algoritmo de Mesclagem

1. Criação de Arrays Temporários

Criamos dois arrays temporários L[] e R[] para armazenar as duas metades a serem mescladas.

2. Cópia dos Dados

Copiamos os elementos das duas metades para os arrays temporários.

3. Mesclagem Ordenada

Comparamos elementos dos dois arrays e os colocamos na posição correta no array original.

4. Elementos Restantes

Copiamos quaisquer elementos restantes de L[] ou R[] para o array original.

Observações Importantes

- **Estabilidade:** O processo de mesclagem mantém a ordem relativa de elementos iguais, tornando o Merge Sort um algoritmo estável.
- **Espaço Extra:** A mesclagem requer espaço auxiliar proporcional ao tamanho dos arrays sendo mesclados (O(n)).
- **Previsibilidade:** O processo de mesclagem sempre realiza no máximo (n-1) comparações para mesclar n elementos.
- **Eficiência:** A mesclagem é um processo linear O(n), mas o Merge Sort completo é O(n log n) devido às divisões recursivas.

Dica: O processo de mesclagem é o que torna o Merge Sort eficiente, pois combina duas listas ordenadas em tempo linear.

Merge Sort: Implementação

Implementação em C++

Vamos implementar o Merge Sort em C++ básico, sem usar recursos avançados como STL ou POO:

```cpp
// Função para imprimir um array
void imprimirArray(int arr[], int tamanho) {
    for (int i = 0; i < tamanho; i++) {
        std::cout << arr[i] << " ";
    }
    std::cout << std::endl;
}

// Função para mesclar duas partes ordenadas do array
void merge(int arr[], int inicio, int meio, int fim) {
    // Tamanho dos subarrays
    int n1 = meio - inicio + 1;
    int n2 = fim - meio;

    // Cria arrays temporários
    int* L = new int[n1];
    int* R = new int[n2];

    // Copia dados para arrays temporários
    for (int i = 0; i < n1; i++) {
        L[i] = arr[inicio + i];
    }
    for (int j = 0; j < n2; j++) {
        R[j] = arr[meio + 1 + j];
    }

    // Mescla os arrays temporários de volta em arr[inicio..fim]
    int i = 0;    // Índice inicial do primeiro subarray
    int j = 0;    // Índice inicial do segundo subarray
    int k = inicio; // Índice inicial do array mesclado

    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    // Copia os elementos restantes de L[], se houver
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    // Copia os elementos restantes de R[], se houver
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }

    // Libera a memória alocada
    delete[] L;
    delete[] R;
}
```

```cpp
// Função principal do Merge Sort
void mergeSort(int arr[], int inicio, int fim) {
    if (inicio < fim) {
        // Encontra o ponto médio
        int meio = inicio + (fim - inicio) / 2;

        // Ordena a primeira e a segunda metade
        mergeSort(arr, inicio, meio);
        mergeSort(arr, meio + 1, fim);

        // Mescla as metades ordenadas
        merge(arr, inicio, meio, fim);
    }
}

// Função principal para testar o Merge Sort
int main() {
    int arr[] = {38, 27, 43, 3, 9, 82, 10};
    int n = sizeof(arr) / sizeof(arr[0]);

    std::cout << "Array original: \n";
    imprimirArray(arr, n);

    mergeSort(arr, 0, n - 1);

    std::cout << "Array ordenado: \n";
    imprimirArray(arr, n);

    return 0;
}
```

Explicação do Código

Função `merge()`

- **Arrays temporários**: Criamos dois arrays L[] e R[] para armazenar as duas metades.
- **Cópia de dados**: Copiamos os elementos das duas metades para os arrays temporários.
- **Mesclagem**: Comparamos elementos dos dois arrays e os colocamos na posição correta.
- **Elementos restantes**: Copiamos quaisquer elementos restantes para o array original.
- **Liberação de memória**: Liberamos a memória alocada para os arrays temporários.

Função `mergeSort()`

- **Caso base**: Se inicio >= fim, o array tem 0 ou 1 elemento e já está ordenado.
- **Divisão**: Encontramos o ponto médio e dividimos o array em duas metades.
- **Recursão**: Ordenamos recursivamente as duas metades.
- **Combinação**: Mesclamos as duas metades ordenadas.

Detalhes de Implementação

- **Alocação dinâmica**: Usamos `new` e `delete` para gerenciar memória.
- **Cálculo do meio**: Usamos `inicio + (fim - inicio) / 2` para evitar overflow.
- **Recursividade**: O algoritmo usa chamadas recursivas para dividir o problema.
- **Estabilidade**: A implementação mantém a ordem relativa de elementos iguais.

Observação: Esta implementação usa alocação dinâmica de memória para os arrays temporários. Em C++ moderno, poderíamos usar `std::vector` para simplificar o gerenciamento de memória, mas estamos usando C++ básico conforme solicitado.

Merge Sort: Análise

Análise de Complexidade

Vamos analisar o Merge Sort em termos de complexidade de tempo, espaço e outras características:

Complexidade de Tempo

Análise do Algoritmo:

- **Divisão:** Dividir o array ao meio leva tempo constante O(1)
- **Recursão:** Duas chamadas recursivas para metades do tamanho n/2
- **Mesclagem:** Mesclar n elementos leva tempo O(n)

Complexidade:

- **Melhor caso:** O(n log n)
- **Caso médio:** O(n log n)
- **Pior caso:** O(n log n)

Equação de Recorrência:

$$T(n) = 2T(n/2) + O(n)$$

Pelo Teorema Mestre, isso resolve para O(n log n)

Outras Características

Complexidade de Espaço:

- **O(n)** - Requer espaço auxiliar proporcional ao tamanho do array
- Arrays temporários são criados durante a mesclagem
- Pilha de recursão também consome espaço O(log n)

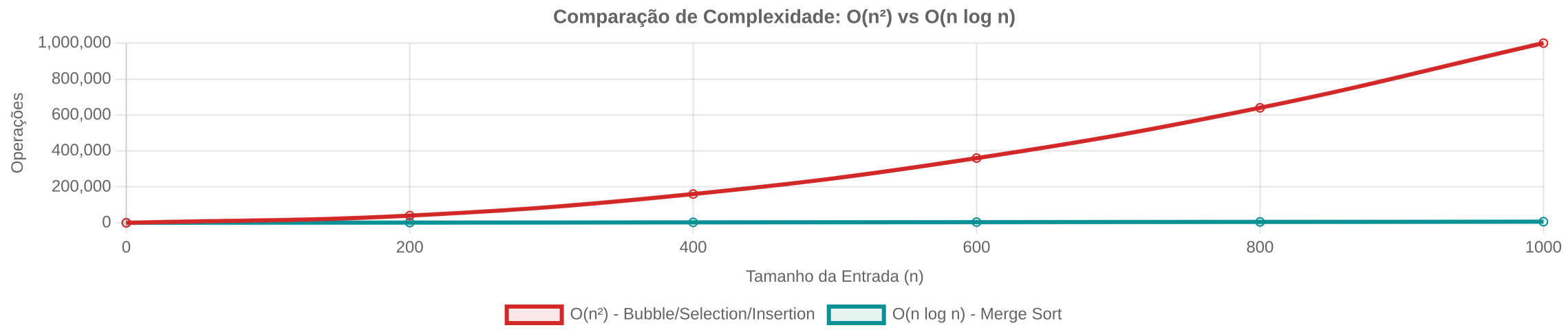
Estabilidade:

- **Estável** - Mantém a ordem relativa de elementos com valores iguais
- Durante a mesclagem, elementos iguais do primeiro subarray são colocados antes dos elementos do segundo subarray

Adaptabilidade:

- **Não adaptativo** - O desempenho não melhora para arrays parcialmente ordenados
- Sempre executa o mesmo número de operações, independentemente da ordem inicial dos elementos

Visualização da Complexidade



Vantagens

- Complexidade garantida O(n log n) em todos os casos
- Estável - mantém a ordem relativa de elementos iguais
- Eficiente para grandes conjuntos de dados
- Bom para ordenação externa (dados que não cabem na memória)
- Paralelizável - diferentes partes podem ser processadas simultaneamente

Desvantagens

- Requer espaço auxiliar O(n) - não é in-place
- Overhead para arrays pequenos
- Não adaptativo - não aproveita ordenação parcial
- Mais complexo de implementar que algoritmos simples
- Alocação e desalocação de memória podem ser custosas

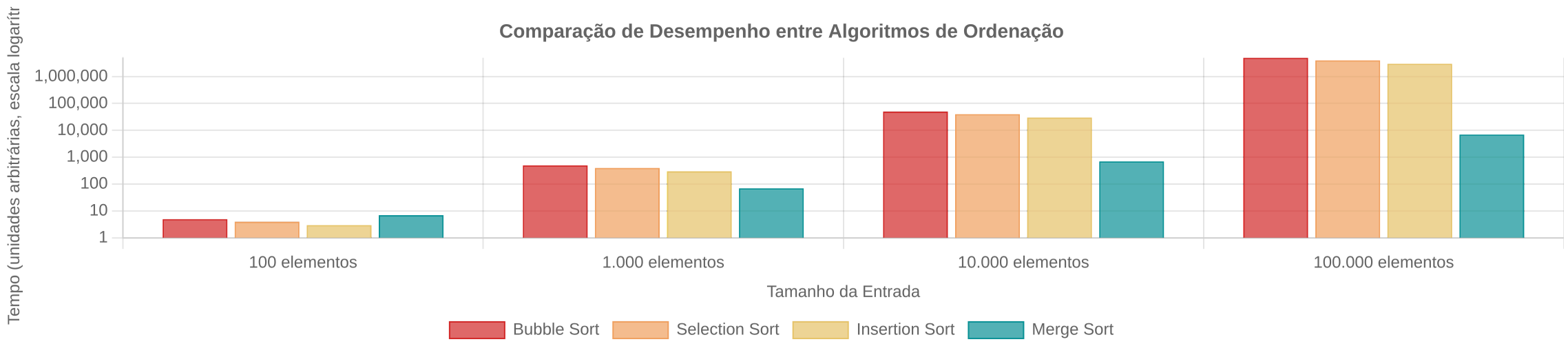
Comparação com Outros Algoritmos

Merge Sort vs. Algoritmos Anteriores

Vamos comparar o Merge Sort com os algoritmos que estudamos anteriormente:

Característica	Bubble Sort	Selection Sort	Insertion Sort	Merge Sort
Melhor Caso	O(n)	O(n²)	O(n)	O(n log n)
Caso Médio	O(n²)	O(n²)	O(n²)	O(n log n)
Pior Caso	O(n²)	O(n²)	O(n²)	O(n log n)
Espaço	O(1)	O(1)	O(1)	O(n)
Estabilidade	Sim	Não	Sim	Sim
Adaptabilidade	Sim	Não	Sim	Não
In-place	Sim	Sim	Sim	Não
Complexidade de Implementação	Simples	Simples	Simples	Moderada

Comparação de Desempenho



Quando Usar Cada Algoritmo

Bubble, Selection, Insertion

Melhor escolha quando:

- O conjunto de dados é **pequeno** (< 50 elementos)
- A **memória é extremamente limitada** (algoritmos in-place)
- A **implementação simples** é mais importante que o desempenho
- Os dados já estão **parcialmente ordenados** (Insertion Sort)

Merge Sort

Melhor escolha quando:

- O conjunto de dados é **grande** (> 1000 elementos)
- Você precisa de **desempenho garantido** O(n log n)
- A **estabilidade** é importante
- Você está trabalhando com **ordenação externa** (dados que não cabem na memória)
- Você pode **paralelizar** o processamento

O Merge Sort representa um salto significativo em eficiência em relação aos algoritmos O(n²), especialmente para grandes conjuntos de dados. No entanto, para conjuntos pequenos, os algoritmos mais simples podem ser mais eficientes devido ao menor overhead.

Variações e Otimizações

Variações do Merge Sort

Existem diversas variações e otimizações do algoritmo Merge Sort que podem melhorar seu desempenho em cenários específicos:

Bottom-up Merge Sort

Implementação iterativa (não recursiva) do Merge Sort que começa mesclando pares de elementos, depois grupos de 4, 8, etc.

Vantagens: Elimina a sobrecarga da recursão e o uso da pilha.

```
void bottomUpMergeSort(int arr[], int n) {
  // Tamanho atual de subarrays a serem mesclados
  for (int size = 1; size < n; size = 2*size) {
    // Ponto de início de cada subarray
    for (int left = 0; left < n-size; left += 2*size) {
      int mid = left + size - 1;
      int right = min(left + 2*size - 1, n-1);
      merge(arr, left, mid, right);
    }
  }
}
```

Natural Merge Sort

Aproveita sequências já ordenadas no array original, identificando "corridas" naturais antes de mesclar.

Vantagens: Adaptativo, mais eficiente para arrays parcialmente ordenados.

- Identifica sequências crescentes no array
- Mescla essas sequências em pares
- Reduz o número de mesclagens necessárias

Merge Sort Híbrido

Usa Insertion Sort para pequenos subarrays (tipicamente < 10-20 elementos) e Merge Sort para o resto.

Vantagens: Reduz o overhead para pequenos arrays onde Insertion Sort é mais eficiente.

```
void hybridMergeSort(int arr[], int inicio, int fim) {
  // Para pequenos arrays, use Insertion Sort
  if (fim - inicio <= 10) {
    insertionSort(arr, inicio, fim);
    return;
  }

  // Para arrays maiores, use Merge Sort
  int meio = inicio + (fim - inicio) / 2;
  hybridMergeSort(arr, inicio, meio);
  hybridMergeSort(arr, meio + 1, fim);
  merge(arr, inicio, meio, fim);
}
```

In-place Merge Sort

Implementação que tenta minimizar o uso de memória auxiliar, realizando a mesclagem no próprio array.

Vantagens: Reduz o uso de memória, mas geralmente com maior complexidade de tempo.

- Usa rotações de blocos ou técnicas de deslocamento
- Complexidade de tempo geralmente O(n log² n)
- Útil quando a memória é extremamente limitada

Otimizações Comuns

1. Evitar Cópias Desnecessárias

Alternar entre dois arrays auxiliares em vez de copiar de volta para o array original a cada mesclagem.

2. Otimização de Cache

Organizar o acesso à memória para maximizar o uso do cache, processando blocos de tamanho adequado.

3. Paralelização

Dividir o trabalho entre múltiplos threads ou processos, especialmente para grandes conjuntos de dados.

4. Verificação de Arrays Já Ordenados

Verificar se os subarrays já estão ordenados antes de mesclar, evitando operações desnecessárias.

Estas variações e otimizações demonstram como um algoritmo clássico pode ser adaptado para diferentes cenários e requisitos. Na prática, bibliotecas de ordenação modernas geralmente implementam versões híbridas e otimizadas destes algoritmos.

Aplicações Práticas

Onde o Merge Sort é Utilizado

O Merge Sort é amplamente utilizado em diversos cenários do mundo real devido às suas características de desempenho e estabilidade:

Ordenação Externa

Ordenação de dados que não cabem completamente na memória principal.

Exemplo: Ordenação de arquivos grandes em sistemas de banco de dados.

Sistemas Distribuídos

Ordenação de dados distribuídos em múltiplos servidores ou nós.

Exemplo: MapReduce e outros frameworks de processamento distribuído.

Análise de Dados

Ordenação de grandes conjuntos de dados para análise estatística.

Exemplo: Cálculo de medianas, percentis e outras estatísticas de ordem.

Bibliotecas Padrão

Implementações de funções de ordenação em bibliotecas de linguagens de programação.

Exemplo: Java usa Timsort (variante do Merge Sort) para Collections.sort().

Algoritmos de Grafos

Ordenação de arestas ou vértices em algoritmos de grafos.

Exemplo: Algoritmo de Kruskal para árvore geradora mínima.

Contagem de Inversões

Contagem de pares de elementos fora de ordem em um array.

Exemplo: Medição de "desordem" em sequências ou similaridade entre rankings.

Caso de Estudo: Ordenação Externa

Um dos usos mais importantes do Merge Sort é na ordenação externa, quando os dados são maiores que a memória disponível:

1. Dividir o arquivo grande em blocos que cabem na memória
2. Ordenar cada bloco individualmente usando qualquer algoritmo eficiente
3. Salvar os blocos ordenados em arquivos temporários
4. Mesclar os arquivos ordenados em pares, criando arquivos maiores ordenados
5. Repetir o processo de mesclagem até obter um único arquivo ordenado

Este processo é fundamental em sistemas de banco de dados, processamento de logs e análise de grandes volumes de dados.

Aplicações Reais

- Ordenação de índices em SGBDs
- Processamento de logs de servidores
- Análise de grandes datasets científicos
- Processamento de transações financeiras

O Merge Sort é especialmente valioso em aplicações onde a estabilidade é importante e onde o desempenho previsível é necessário, independentemente da distribuição inicial dos dados.

Exercícios Propostos

Atividades Práticas

Para consolidar o aprendizado sobre o Merge Sort, propomos os seguintes exercícios:

1 Implementação e Teste

Implemente o Merge Sort em C++ e compare seu desempenho com os algoritmos vistos anteriormente (Bubble, Selection e Insertion Sort) usando diferentes tamanhos de entrada:

- 100 elementos
- 1.000 elementos
- 10.000 elementos

Dica: Use a função `clock()` da biblioteca `ctime` para medir o tempo de execução.

2 Implementação Bottom-up

Implemente a versão iterativa (bottom-up) do Merge Sort e compare seu desempenho com a versão recursiva.

```
// Estrutura básica para o Merge Sort Bottom-up
void bottomUpMergeSort(int arr[], int n) {
    // Tamanho atual de subarrays a serem mesclados
    for (int size = 1; size < n; size = 2*size) {
        // Ponto de início de cada subarray
        for (int left = 0; left < n-size; left += 2*size) {
            int mid = left + size - 1;
            int right = min(left + 2*size - 1, n-1);
            // Implemente a função merge aqui
            // merge(arr, left, mid, right);
        }
    }
}
```

Dica: Compare o desempenho e o uso de memória das duas implementações.

3 Contagem de Inversões

Modifique o algoritmo Merge Sort para contar o número de inversões em um array (pares de elementos que estão fora de ordem).

```
// Função para contar inversões usando Merge Sort
int countInversions(int arr[], int n) {
    // Implemente esta função usando o conceito
    // de Merge Sort para contar inversões
    // durante o processo de mesclagem

    // Uma inversão ocorre quando arr[i] > arr[j] e i < j

    return inversions; // Retorna o número total de inversões
}
```

Dica: Durante a mesclagem, quando um elemento do subarray direito é escolhido antes de um elemento do subarray esquerdo, isso indica inversões.

4 Merge Sort Híbrido

Implemente uma versão híbrida do Merge Sort que usa Insertion Sort para subarrays pequenos (menores que um limiar K).

```
// Função para o Merge Sort Híbrido
void hybridMergeSort(int arr[], int inicio, int fim, int K) {
    // Se o tamanho do subarray for menor que K,
    // use Insertion Sort
    if (fim - inicio + 1 <= K) {
        insertionSort(arr, inicio, fim);
        return;
    }

    // Caso contrário, use Merge Sort normal
    int meio = inicio + (fim - inicio) / 2;
    hybridMergeSort(arr, inicio, meio, K);
    hybridMergeSort(arr, meio + 1, fim, K);
    merge(arr, inicio, meio, fim);
}
```

Dica: Experimente diferentes valores de K (5, 10, 15, 20) e compare o desempenho.

Desafio Extra

Implemente uma simulação de ordenação externa usando o Merge Sort:

1. Crie um arquivo grande com números aleatórios (pelo menos 1 milhão de números)
2. Divida o arquivo em blocos menores que cabem na memória
3. Ordene cada bloco usando Merge Sort e salve em arquivos temporários
4. Mescle os arquivos temporários em pares até obter um único arquivo ordenado

Pergunta: Como o tamanho dos blocos afeta o desempenho geral do algoritmo? Qual seria o tamanho ideal de bloco para seu sistema?

Entrega dos Exercícios

- Arquivos .cpp com as implementações solicitadas
- Relatório com análises de desempenho e comparações
- Gráficos comparativos entre os diferentes algoritmos
- Data de entrega: Próxima aula

Próximas Aulas

O Que Veremos a Seguir

Nas próximas aulas, continuaremos nossa jornada pelos algoritmos de ordenação, explorando métodos baseados em distribuição:

Algoritmos de Ordenação por Distribuição

Aulas 7-8: Counting Sort

Mudaremos nossa abordagem para explorar algoritmos que não dependem de comparações entre elementos, mas sim da distribuição dos valores em faixas ou buckets.

Principais Características

- Não baseado em comparações
- Complexidade linear O(n+k) em casos favoráveis
- Ideal para conjuntos com faixa de valores conhecida
- Estável (mantém a ordem relativa de elementos iguais)

Tópicos que Abordaremos

- Conceito de ordenação por contagem
- Implementação do Counting Sort
- Análise de complexidade e limitações
- Aplicações práticas e otimizações

Cronograma das Próximas Aulas

7-8 Counting Sort

- Conceito e funcionamento
- Implementação em C++
- Análise de complexidade
- Exercícios práticos

9-10 Radix Sort

- Ordenação por dígitos
- Implementação em C++
- Análise de complexidade
- Comparação com outros algoritmos

11-12 Revisão e Projeto Final

- Comparação geral dos algoritmos
- Quando usar cada algoritmo
- Projeto prático integrador
- Avaliação final

Preparação para a Próxima Aula

Para aproveitar melhor a próxima aula sobre Counting Sort, recomendamos:

- Revisar os conceitos de arrays e contagem de frequência
- Concluir os exercícios propostos sobre Merge Sort
- Pensar em cenários onde conhecemos o intervalo de valores possíveis