Guia Completo: Implementando o Algoritmo Quicksort em C Puro

Introdução

O Quicksort é um dos algoritmos de ordenação mais eficientes e amplamente utilizados. Desenvolvido por Tony Hoare em 1960, ele é baseado na estratégia de 'dividir para conquistar', o que o torna particularmente rápido para grandes conjuntos de dados. Sua popularidade deriva de sua velocidade em casos médios e de sua implementação relativamente simples, embora a compreensão completa de sua recursividade e do processo de particionamento exija atenção.

Este documento tem como objetivo fornecer um guia abrangente para a implementação do Quicksort em C puro, sem o uso de bibliotecas de ordenação pré-existentes. Abordaremos a teoria por trás do algoritmo, detalharemos cada componente de sua implementação, forneceremos exemplos de código e discutiremos sua complexidade e características.

1. A Estratégia 'Dividir para Conquistar'

O Quicksort segue a filosofia 'dividir para conquistar', que consiste em três etapas principais:

1. **Dividir (Divide):** O problema é dividido em subproblemas menores e independentes. No Quicksort, isso envolve a seleção de um 'pivô' e a reorganização do array de forma que elementos menores que o pivô fiquem de um lado e elementos maiores fiquem do outro. O pivô, por sua vez, é colocado em sua posição final correta.
2. **Conquistar (Conquer):** Os subproblemas são resolvidos recursivamente. O Quicksort é chamado para ordenar os sub-arrays à esquerda e à direita do pivô.
3. **Combinar (Combine):** As soluções dos subproblemas são combinadas para resolver o problema original. No Quicksort, a etapa de combinação é trivial, pois o array já está ordenado após o particionamento e as chamadas recursivas.

2. Função Auxiliar: `swap`

Antes de mergulharmos no coração do Quicksort, que é a função de particionamento, precisamos de uma função auxiliar simples para trocar a posição de dois elementos em um array. Esta função será fundamental para reorganizar os elementos durante o processo de particionamento.

A função `swap` recebe dois ponteiros para inteiros e troca os valores para os quais esses ponteiros apontam. Isso garante que as modificações sejam feitas diretamente nos elementos do array original, e não em cópias locais.

Plain Text

```
void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

Explicação:

- `int* a` e `int* b`: Declaram `a` e `b` como ponteiros para inteiros. Isso significa que eles armazenarão endereços de memória onde valores inteiros estão guardados.
- `int temp = *a;`: Cria uma variável temporária `temp` e armazena nela o valor apontado por `a` (ou seja, o valor que está no endereço de memória de `a`).
- `*a = *b;`: Atribui ao endereço de memória de `a` o valor apontado por `b`.
- `*b = temp;`: Atribui ao endereço de memória de `b` o valor que estava originalmente em `a` (e que foi salvo em `temp`).

Com esta função, podemos facilmente trocar a posição de quaisquer dois elementos em nosso array.

3. A Função de Particionamento (`partition`)

A função de particionamento é o coração do algoritmo Quicksort. Sua principal responsabilidade é rearranjar os elementos de um sub-array de forma que todos os elementos menores ou iguais a um pivô escolhido fiquem à sua esquerda, e todos os elementos maiores que o pivô fiquem à sua direita. Ao final do processo, o pivô estará em sua posição final correta no array ordenado.

Existem várias estratégias para escolher o pivô (primeiro elemento, último elemento, elemento do meio, elemento aleatório). Para simplificar a implementação e manter a didática, neste guia, utilizaremos o **último elemento do sub-array como pivô**.

Detalhamento do Processo de Particionamento

Vamos descrever o algoritmo de particionamento de Hoare (embora a implementação abaixo seja mais próxima da de Lomuto, que é mais simples de entender para iniciantes ao usar o último elemento como pivô):

1. **Escolha do Pivô:** O elemento na posição `high` (o último elemento do sub-array atual) é selecionado como pivô.
2. **Inicialização do Índice `i`:** Um índice `i` é inicializado como `low - 1`. Este `i` será o limite superior da seção de elementos menores ou iguais ao pivô.
3. **Iteração pelos Elementos:** Um loop `j` percorre os elementos do sub-array de `low` até `high - 1` (excluindo o pivô).
4. **Comparação e Troca:** Para cada elemento `vetor[j]`:
* Se `vetor[j]` for menor ou igual ao pivô, significa que ele pertence à seção dos elementos menores. Nesse caso, incrementamos `i` e trocamos `vetor[i]` com `vetor[j]`. Isso efetivamente move o elemento menor para a parte inicial do sub-array.
5. **Posicionamento Final do Pivô:** Após o loop, todos os elementos menores ou iguais ao pivô estão nas posições de `low` até `i`. O pivô (que está em `vetor[high]`) é então trocado com o elemento em `vetor[i + 1]`. Isso coloca o pivô em sua posição final correta, com todos os elementos menores à sua esquerda e todos os maiores à sua direita.
6. **Retorno do Índice do Pivô:** A função retorna o índice `i + 1`, que é a posição final do pivô.

Código da Função `partition`

Plain Text

```
int partition(int vetor[], int low, int high) {
    int pivo = vetor[high]; // Escolhe o último elemento como pivô
    int i = (low - 1);      // Índice do menor elemento

    for (int j = low; j <= high - 1; j++) {
        // Se o elemento atual é menor ou igual ao pivô
        if (vetor[j] <= pivo) {
            i++; // Incrementa o índice do menor elemento
            swap(&vetor[i], &vetor[j]); // Troca o elemento atual com o
elemento no índice i
        }
    }
    // Coloca o pivô em sua posição correta
    swap(&vetor[i + 1], &vetor[high]);
    return (i + 1); // Retorna o índice da posição final do pivô
}
```

Exemplo Ilustrativo do Particionamento:

Vamos considerar o array `[10, 80, 30, 90, 40, 50, 70]` com `low = 0` e `high = 6`.

- **Pivô:** `70` (elemento em `vetor[6]`).
- **`i`:** `-1`

Passo	j	vetor[j]	pivo	vetor[j] <= pivo ?	i (após incremento)	Array (após swap)	Observações
Início			70		-1	[10, 80, 30, 90, 40, 50, 70]	pivo = 70
1	0	10	70	Sim	0	[10, 80, 30, 90, 40, 50, 70]	10 é menor que 70. swap(vetor[0], vetor[0]) (sem efeito)
2	1	80	70	Não	0	[10, 80, 30, 90, 40, 50, 70]	80 é maior que 70. Nada acontece.
3	2	30	70	Sim	1	[10, 30, 80, 90, 40, 50, 70]	30 é menor que 70. swap(vetor[1], vetor[2]) (80 e 30 trocam)
4	3	90	70	Não	1	[10, 30, 80, 90, 40, 50, 70]	90 é maior que 70. Nada acontece.
5	4	40	70	Sim	2	[10, 30, 40, 90, 80, 50, 70]	40 é menor que 70. swap(vetor[2], vetor[4]) (80 e 40 trocam)
6	5	50	70	Sim	3	[10, 30, 40, 50, 80, 90, 70]	50 é menor que 70. swap(vetor[3], vetor[5]) (90 e 50 trocam)
Final					3	[10, 30, 40, 50, 80, 90, 70]	Loop j terminou. i é 3.

Após o loop, antes de posicionar o pivô:

Array: [10, 30, 40, 50, 80, 90, 70]
i = 3
i + 1 = 4
vetor[i + 1] é vetor[4] que é 80.
vetor[high] é vetor[6] que é 70 (o pivô).

Troca final: `swap(&vetor[i + 1], &vetor[high])` ou `swap(&vetor[4], &vetor[6])`

Array após a troca: `[10, 30, 40, 50, 70, 90, 80]`

Resultado: O pivô `70` está agora na posição `4`. Todos os elementos à sua esquerda (`10, 30, 40, 50`) são menores ou iguais a `70`. Todos os elementos à sua direita (`90, 80`) são maiores que `70`. A função `partition` retornaria `4`.

4. A Função Principal do Quicksort (`quickSort`)

A função `quickSort` é a orquestradora do algoritmo. Ela utiliza a função `partition` para dividir o array em subproblemas menores e, em seguida, chama a si mesma recursivamente para ordenar esses subproblemas. Este é o cerne da estratégia de 'dividir para conquistar'.

Detalhamento da Função `quickSort`

1. **Parâmetros:** A função recebe o array (`vetor[]`), o índice inicial (`low`) e o índice final (`high`) do sub-array que deve ser ordenado.
2. **Condição de Parada (Base Case):** A recursão precisa de uma condição de parada para evitar um loop infinito. No Quicksort, a condição de parada ocorre quando `low` é maior ou igual a `high`. Isso significa que o sub-array atual tem zero ou um elemento, e um array com zero ou um elemento é, por definição, já ordenado. Portanto, não há mais trabalho a ser feito para esse sub-array.
3. **Chamada de Particionamento:** Se `low < high` (ou seja, o sub-array tem dois ou mais elementos), a função chama `partition(vetor, low, high)`. Esta chamada retorna o `pi` (índice de particionamento), que é a posição final correta do pivô no array.
4. **Chamadas Recursivas:** Após o particionamento, o pivô está em sua posição final. Agora, o problema original foi dividido em dois subproblemas independentes:
   - Ordenar o sub-array à esquerda do pivô: `quickSort(vetor, low, pi - 1)`.
   - Ordenar o sub-array à direita do pivô: `quickSort(vetor, pi + 1, high)`.

Essas chamadas recursivas continuarão dividindo e conquistando até que todos os sub-arrays atinjam a condição de parada, momento em que o array completo estará ordenado.

Código da Função `quickSort`

Plain Text

```
void quickSort(int vetor[], int low, int high) {
    // Condição de parada da recursão: se o sub-array tem 0 ou 1 elemento,
ele já está ordenado.
```

```
    if (low < high) {
        // pi é o índice de particionamento, vetor[pi] está agora no lugar
certo
        int pi = partition(vetor, low, high);

        // Ordena os elementos separadamente antes e depois da partição
        quickSort(vetor, low, pi - 1);  // Chama Quicksort para o sub-array
à esquerda do pivô
        quickSort(vetor, pi + 1, high); // Chama Quicksort para o sub-array
à direita do pivô
    }
}
```

5. Exemplo Completo de Implementação em C

Para ilustrar como todas as partes se encaixam, aqui está um programa C completo que implementa o Quicksort e o utiliza para ordenar um array de inteiros. Este exemplo inclui as funções `swap`, `partition`, `quickSort` e uma função `printArray` para exibir o conteúdo do array.

Plain Text

```
#include <stdio.h>

// Função auxiliar para trocar dois elementos
void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Função de particionamento (implementação de Lomuto, usando o último
elemento como pivô)
int partition(int vetor[], int low, int high) {
    int pivo = vetor[high]; // Escolhe o último elemento como pivô
    int i = (low - 1);      // Índice do menor elemento

    for (int j = low; j <= high - 1; j++) {
        // Se o elemento atual é menor ou igual ao pivô
        if (vetor[j] <= pivo) {
            i++; // Incrementa o índice do menor elemento
            swap(&vetor[i], &vetor[j]); // Troca o elemento atual com o
elemento no índice i
        }
    }
    // Coloca o pivô em sua posição correta
```

```
    swap(&vetor[i + 1], &vetor[high]);
    return (i + 1); // Retorna o índice da posição final do pivô
}

// Função principal que implementa o Quicksort
void quickSort(int vetor[], int low, int high) {
    // Condição de parada da recursão: se o sub-array tem 0 ou 1 elemento,
ele já está ordenado.
    if (low < high) {
        // pi é o índice de particionamento, vetor[pi] está agora no lugar
certo
        int pi = partition(vetor, low, high);

        // Ordena os elementos separadamente antes e depois da partição
        quickSort(vetor, low, pi - 1);  // Chama Quicksort para o sub-array
à esquerda do pivô
        quickSort(vetor, pi + 1, high); // Chama Quicksort para o sub-array
à direita do pivô
    }
}

// Função para imprimir um array
void printArray(int vetor[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", vetor[i]);
    }
    printf("\n");
}

// Função principal (main) para testar o Quicksort
int main() {
    int arr[] = {10, 7, 8, 9, 1, 5};
    int n = sizeof(arr) / sizeof(arr[0]);
    printf("Array original: ");
    printArray(arr, n);

    quickSort(arr, 0, n - 1);

    printf("Array ordenado: ");
    printArray(arr, n);

    // Teste com outro array
    int arr2[] = {50, 20, 10, 40, 30};
    int n2 = sizeof(arr2) / sizeof(arr2[0]);
    printf("\nArray original 2: ");
    printArray(arr2, n2);

    quickSort(arr2, 0, n2 - 1);
```

```
    printf("Array ordenado 2: ");
    printArray(arr2, n2);

    return 0;
}
```

6. Complexidade de Tempo e Espaço

A eficiência de um algoritmo é geralmente medida por sua complexidade de tempo (quantas operações ele realiza) e complexidade de espaço (quanto de memória ele utiliza). O Quicksort, embora muito eficiente na prática, possui diferentes complexidades dependendo do cenário.

Complexidade de Tempo

- **Melhor Caso (Best Case): O(n log n)**
  Ocorre quando o pivô escolhido sempre divide o array em duas partes aproximadamente iguais. Isso leva a um balanceamento perfeito das chamadas recursivas, resultando em um desempenho logarítmico na profundidade da recursão e linear no trabalho por nível.
- **Caso Médio (Average Case): O(n log n)**
  Na maioria das situações práticas, o Quicksort se comporta de forma muito próxima ao seu melhor caso. Mesmo com divisões ligeiramente desbalanceadas, o desempenho médio permanece excelente. É por isso que ele é tão popular.
- **Pior Caso (Worst Case): O(n^2)**
  O pior caso ocorre quando o pivô escolhido sempre resulta em uma partição extremamente desbalanceada, por exemplo, quando o array já está ordenado (ou inversamente ordenado) e o pivô é sempre o menor (ou maior) elemento. Nesse cenário, o Quicksort degenera para um algoritmo semelhante ao Bubble Sort, onde cada particionamento reduz o problema em apenas um elemento, levando a n níveis de recursão e n operações por nível.

Complexidade de Espaço

- **Complexidade de Espaço (Space Complexity): O(log n) no caso médio, O(n) no pior caso.**
  A complexidade de espaço do Quicksort é determinada pela pilha de chamadas recursivas. No caso médio, onde as partições são balanceadas, a profundidade da recursão é $\log n$, resultando em $O(\log n)$ espaço. No pior caso, quando as partições são desbalanceadas, a profundidade da recursão pode ser n, levando a $O(n)$ espaço. Isso

pode ser um problema para arrays muito grandes, pois pode causar um estouro de pilha (stack overflow).

Comparação com Outros Algoritmos de Ordenação

A tabela abaixo resume a complexidade de tempo e espaço do Quicksort em comparação com outros algoritmos de ordenação comuns:

Algoritmo de Ordenação	Complexidade de Tempo (Melhor)	Complexidade de Tempo (Médio)	Complexidade de Tempo (Pior)	Complexidade de Espaço
Quicksort	O(n log n)	O(n log n)	O(n^2)	O(log n) (médio), O(n) (pior)
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)
Heap Sort	O(n log n)	O(n log n)	O(n log n)	O(1)
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)

É importante notar que, apesar do pior caso de `O(n^2)`, o Quicksort é geralmente mais rápido na prática do que outros algoritmos `O(n log n)` como Merge Sort ou Heap Sort devido à sua constante de fator menor e melhor desempenho de cache (localidade de referência).

7. Adaptando o Quicksort para Ordem Decrescente

O Quicksort que implementamos ordena os elementos em ordem crescente. No entanto, é muito comum a necessidade de ordenar em ordem decrescente. Existem duas abordagens principais para conseguir isso com o Quicksort:

Abordagem 1: Modificar a Função de Comparação no `partition`

Esta é a abordagem mais direta e elegante. A lógica de ordenação (crescente ou decrescente) reside na comparação feita dentro da função `partition`. Para ordenar em ordem decrescente, basta inverter a condição de comparação.

No nosso `partition` original, usamos:

Plain Text

```
if (vetor[j] <= pivo) {
    // ... troca ...
}
```

Esta condição garante que elementos menores ou iguais ao pivô sejam movidos para a esquerda. Para ordenar em ordem decrescente, queremos que elementos *maiores ou iguais* ao pivô sejam movidos para a esquerda (ou seja, para as posições iniciais do array, que se tornarão o

início do array ordenado decrescentemente).

Assim, a modificação seria:

Plain Text

```
// Modificação na função partition para ordem decrescente
int partition_decrescente(int vetor[], int low, int high) {
    int pivo = vetor[high]; // Escolhe o último elemento como pivô
    int i = (low - 1);      // Índice do menor elemento

    for (int j = low; j <= high - 1; j++) {
        // Se o elemento atual é MAIOR ou igual ao pivô
        if (vetor[j] >= pivo) { // AQUI ESTÁ A MUDANÇA
            i++; // Incrementa o índice do maior elemento
            swap(&vetor[i], &vetor[j]); // Troca o elemento atual com o
elemento no índice i
        }
    }
    // Coloca o pivô em sua posição correta
    swap(&vetor[i + 1], &vetor[high]);
    return (i + 1); // Retorna o índice da posição final do pivô
}

// A função principal que implementa o Quicksort para ordem decrescente
void quickSort_decrescente(int vetor[], int low, int high) {
    if (low < high) {
        int pi = partition_decrescente(vetor, low, high);
        quickSort_decrescente(vetor, low, pi - 1);
        quickSort_decrescente(vetor, pi + 1, high);
    }
}
```

Com essa pequena alteração na condição de comparação dentro da função `partition`, o Quicksort passará a ordenar o array em ordem decrescente.

Abordagem 2: Ordenar Crescente e Inverter a Impressão

Uma alternativa mais simples, especialmente se você já tem a implementação do Quicksort crescente funcionando e não quer modificá-la, é ordenar o array em ordem crescente e, na hora de imprimir, percorrê-lo do último elemento para o primeiro.

Esta abordagem não modifica o algoritmo de ordenação em si, apenas a forma como o resultado é apresentado. É útil quando a ordenação decrescente é uma necessidade apenas na saída, e não na estrutura interna dos dados.

Plain Text

```
// Exemplo de impressão decrescente após ordenação crescente
void printArrayDecrescente(int vetor[], int size) {
    for (int i = size - 1; i >= 0; i--) {
        printf("%d ", vetor[i]);
    }
    printf("\n");
}

// Uso:
// int arr[] = {10, 7, 8, 9, 1, 5};
// int n = sizeof(arr) / sizeof(arr[0]);
// quickSort(arr, 0, n - 1); // Ordena crescentemente
// printArrayDecrescente(arr, n); // Imprime decrescentemente
```

8. Considerações Finais e Otimizações

O Quicksort é um algoritmo poderoso, mas algumas considerações podem otimizar seu desempenho e robustez:

- **Escolha do Pivô:** A escolha do pivô é crucial para o desempenho. Escolher o pivô aleatoriamente ou usar a

estratégia da 'mediana de três' (escolher a mediana entre o primeiro, o último e o elemento do meio) pode ajudar a mitigar o risco do pior caso, tornando o desempenho médio mais consistente.

- **Arrays Pequenos:** Para sub-arrays muito pequenos (por exemplo, com menos de 10-20 elementos), o Quicksort pode ser menos eficiente que algoritmos mais simples como o Insertion Sort devido à sobrecarga da recursão. Uma otimização comum é usar o Insertion Sort para ordenar esses pequenos sub-arrays após o Quicksort ter dividido o array principal até um certo tamanho.
- **Otimização de Recursão de Cauda (Tail Recursion Optimization):** Em algumas linguagens e compiladores, a recursão de cauda pode ser otimizada para evitar o

estouro de pilha no pior caso. No Quicksort, isso pode ser aplicado chamando recursivamente o menor dos dois sub-arrays primeiro e transformando a chamada para o maior sub-array em um loop iterativo, reduzindo a profundidade máxima da pilha para `O(log n)`.

- **Estabilidade:** O Quicksort, em sua implementação padrão, **não é um algoritmo de ordenação estável**. Isso significa que a ordem relativa de elementos com valores iguais pode não ser preservada após a ordenação. Se a estabilidade for um requisito, outros algoritmos como Merge Sort podem ser mais apropriados.

9. Exercícios Propostos

Para solidificar seu entendimento sobre o Quicksort, tente resolver os seguintes exercícios:

1. **Implemente o Quicksort com Mediana de Três:** Modifique a função `partition` para escolher o pivô como a mediana entre o primeiro, o último e o elemento do meio do sub-array. Isso ajuda a evitar o pior caso.
2. **Quicksort para Strings:** Adapte o algoritmo Quicksort para ordenar um array de strings (cadeias de caracteres) em ordem alfabética. Lembre-se de usar funções de comparação de strings (como `strcmp` da biblioteca `<string.h>`).
3. **Quicksort para Estruturas Personalizadas:** Crie uma estrutura `Pessoa` com campos `nome` (string) e `idade` (inteiro). Implemente o Quicksort para ordenar um array de `Pessoa`s primeiramente por `idade` (crescente) e, em caso de idades iguais, por `nome` (alfabética crescente).
4. **Otimização para Pequenos Arrays:** Modifique sua implementação do Quicksort para que, quando o tamanho do sub-array for menor que um determinado limite (por exemplo, 10), ele use o Insertion Sort em vez de continuar a recursão do Quicksort.
5. **Problema do Beecrowd 1259 (Pares e Ímpares):** Volte ao problema original que motivou este guia. Utilize o conhecimento adquirido para implementar a solução, separando os números em pares e ímpares, ordenando os pares crescentemente com Quicksort e os ímpares decrescentemente (usando uma das abordagens discutidas) e imprimindo-os na ordem correta.

Referências

[1] Hoare, C. A. R. (1961). Algorithm 64: Quicksort. *Communications of the ACM*, 4(7), 321. Disponível em: https://dl.acm.org/doi/10.1145/366622.366634

[2] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press. (Capítulo sobre Quicksort)

[3] GeeksforGeeks. (n.d.). *QuickSort*. Disponível em: https://www.geeksforgeeks.org/quick-sort/

[4] Wikipedia. (n.d.). *Quicksort*. Disponível em: https://en.wikipedia.org/wiki/Quicksort

---

Autor: Manus AI
Data: 16 de Agosto de 2025