Análise de Algoritmos de Ordenação

Yan Gonçalves Santana - RGA: 2022.1907.0570

FACOM – Universidade Federal de Mato Grosso do Sul (UFMS) – Campo Grande – MS – Brazil

1. Introdução

Este relatório analisa o desempenho dos seguintes algoritmos de ordenação implementados em C para ordenar os arrays na ordem decrescente:

- Selection Sort
- Insertion Sort
- Merge Sort
- Quick Sort

Os testes foram realizados para entradas aleatórias de tamanho 10^4, 10^5 e 10^6 em um processador Ryzen 5 5600X.

2. Descrição dos Algoritmos

- Selection Sort:

O Selection Sort recursivo percorre o array procurando o maior elemento e colocando-o na última posição. Em seguida, chama-se recursivamente a função para ordenar o restante do array.

O algoritmo consiste em:

1. Percorrer o array da primeira posição (índice i) até a penúltima.
2. Considerar o primeiro elemento como o maior valor atual.
3. Percorrer o restante do array da posição i+1 até o final.
4. Se encontrar um elemento maior que o atual, este se torna o novo maior valor atual.
5. Ao final do loop interno, o maior valor atual será o maior elemento no subarray de i até o final.
6. Trocar o maior elemento encontrado de lugar com o elemento na posição i.
7. Incrementar i e repetir o processo para os próximos elementos.

Dessa forma, o Selection Sort vai colocando iterativamente o maior elemento da parte não ordenada do array na próxima posição, até o array estar completamente ordenado em ordem decrescente.

A implementação recursiva segue o mesmo princípio, porém ao invés de usar loop, chama recursivamente a função, como mostra abaixo na imagem, passando o próximo índice i até chegar ao final do array.

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

// Selection Sort Recursivo
void selectionSort(int vet[], int n, int i) {
    if (i == n - 1)
        return; // Caso base: quando i chega ao final do array

    int aux = i;
    for (int j = i + 1; j < n; j++) {
        if (vet[j] > vet[aux]) {
            aux = j;
        }
    }

    if (aux != i)
        troca(&vet[i], &vet[aux]);

    // Chama recursivamente para o próximo elemento
    selectionSort(vet, n, i + 1);
}
```

- **Insertion Sort:**

O Insertion Sort recursivo percorre o array da direita para a esquerda e à medida que retrocede, insere cada elemento na posição correta, de forma que a parte da direita do array fique ordenada. Em seguida, chama-se recursivamente a função para o restante do array.

Seu funcionamento é:

1. Percorre o array da direita para a esquerda, com um índice i iniciando no último elemento.
2. Define o elemento na posição i como a chave a ser inserida.
3. Percorre os elementos ordenados à esquerda de i, do maior ao menor.
4. Se o elemento da esquerda for menor que a chave, move esse elemento para a direita, abrindo espaço para inserir a chave.
5. Quando encontra um elemento maior ou igual à chave, insere a chave nessa posição.

6. Decrementa i e repete o processo para inserir cada elemento na posição correta em relação aos elementos já ordenados à sua esquerda.

A implementação recursiva segue o mesmo princípio, porém ao invés de loop, chama recursivamente a função decrementando o índice i até chegar no início do array, como mostrado na imagem abaixo.

Dessa forma, o Insertion Sort vai inserindo ordenadamente cada elemento em relação à parte já ordenada do array, até o array estar completamente ordenado em ordem decrescente.

```c
void insertionSort(int vet[], int n, int i) {
    if (i <= 0)
        return; // Caso base: quando i chega a 0 ou menos, o array está ordenado

    int chave = vet[i];
    int j = i - 1;

    while (j >= 0 && vet[j] < chave) {
        vet[j + 1] = vet[j];
        j--;
    }

    vet[j + 1] = chave;

    // Chama recursivamente para o próximo elemento
    insertionSort(vet, n, i - 1);
}
```

- **Merge Sort:**

O Merge Sort utiliza a estratégia de divisão e conquista de forma recursiva para ordenar o array em ordem decrescente, sendo a função intercala responsável por mesclar (intercalar) dois subarrays já ordenados em um único subarray ordenado. Ela recebe como parâmetros o array original, o índice inicial e final de cada um dos dois subarrays a serem mesclados.

O processo feito pela intercala() é:

1. Alocar dois arrays auxiliares L e R para guardar os elementos dos dois subarrays a serem mesclados.
2. Copiar os elementos dos subarrays original para L e R.
3. Percorrer L e R comparando seus elementos e inserindo sempre o maior na próxima posição do array original, deixando assim os elementos em ordem decrescente.

4. Quando um dos arrays L ou R termina, copia-se o restante do outro para o array original.
5. Liberar a memória de L e R.

```c
// intercala - Merge Sort
void intercala(int vet[], int inicio, int meio, int fim) {
    int n1 = meio - inicio + 1;
    int n2 = fim - meio;

    // L e R é alocado dinamicamente no heap usando a função malloc.
    int *L = (int *)malloc(sizeof(int) * n1);
    int *R = (int *)malloc(sizeof(int) * n2);

    for (int i = 0; i < n1; i++)
        L[i] = vet[inicio + i];

    for (int j = 0; j < n2; j++)
        R[j] = vet[meio + 1 + j];

    int i = 0, j = 0, k = inicio;

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

    while (i < n1) {
        vet[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        vet[k] = R[j];
        j++;
        k++;
    }

    // Libera a memória alocada
    free(L);
    free(R);
}
```

Já a função mergeSort() é responsável por fazer a recursão e as chamadas para a intercala(). Ela recebe o array original, índice inicial e final, e segue os seguintes passos:

1. Se o índice inicial é menor que o final, significa que há mais de um elemento.
2. Calcula o ponto do meio entre início e fim.
3. Chama mergeSort() recursivamente para a primeira metade.
4. Chama mergeSort() recursivamente para a segunda metade.
5. Chama intercala() para mesclar as duas metades.

Assim, a mergeSort() vai dividindo o array ao meio recursivamente até sobrarem arrays de 1 elemento, e na volta das chamadas recursivas vai mesclando cada parte por meio da intercala().

```c
// Merge Sort
void mergeSort(int vet[], int inicio, int fim) {
    if (inicio < fim) {
        int meio = inicio + (fim - inicio) / 2;
        mergeSort(vet, inicio, meio);
        mergeSort(vet, meio + 1, fim);
        intercala(vet, inicio, meio, fim);
    }
}
```

- **Quick Sort:**

O Quick Sort também segue a abordagem de divisão e conquista. Escolhe um pivô e particiona o array colocando os menores elementos antes do pivô e os maiores depois. Sendo a função separa responsável por particionar o array em torno de um pivô. Ela recebe o array, índices inicial e final, e realiza os seguintes passos:

1. Escolhe o pivô, que inicialmente é o elemento na posição inicial.
2. Inicializa o índice i como inicial-1 e j como final+1.
3. Percorre o array do início ao fim:

   3.1 Compara cada elemento com o pivô. Se for menor, incrementa i e troca o elemento na posição i com o elemento avaliado.

   3.2 Se o elemento for maior que o pivô, decrementa j e troca o elemento com o da posição j.

4. Quando i e j se cruzam, significa que os menores foram colocados antes e os maiores depois do pivô.
5. Troca o pivô (elemento inicial) para a posição final de j.
6. Retorna a posição final j, dividindo o array em partições menores e maiores que o pivô.

```c
// Separa - Quick Sort
int separa(int vet[], int inicio, int fim) {
    int x = vet[inicio];
    int i = inicio - 1;
    int j = fim + 1;

    while (1) {
        do {
            j--;
        } while (vet[j] < x);

        do {
            i++;
        } while (vet[i] > x);

        if (i >= j)
            return j;

        troca(&vet[i], &vet[j]);
    }
}
```

Já a função quickSort implementa a lógica principal do algoritmo Quick Sort de forma recursiva. Ela recebe como parâmetros o array a ser ordenado, o índice inicial e o índice final.

Seu funcionamento consiste em:

1. Verificar se o índice inicial é menor que o final, ou seja, se há mais de um elemento no subarray.
2. Chamar a função separa(), passando o array, índice inicial e final. A separa irá particionar o array e retornar o índice do pivô.
3. Chamar quickSort recursivamente para a partição à esquerda do pivô, passando o índice inicial e o índice retornado pela separa - 1.
4. Chamar quickSort recursivamente para a partição à direita do pivô, passando o índice retornado pela separa + 1 e o índice final.

Dessa forma, a quickSort() vai recursivamente chamando a si própria para continuar particionando e ordenando as partições menores do array em relação a um pivô, até sobrarem partições de um único elemento.

A separa() é responsável por escolher o pivô, colocar os elementos menores antes dele e os maiores depois, retornando o índice final onde o pivô foi colocado. Assim, a quickSort() consegue particionar e ordenar todas as partições apenas chamando a separa() e a si própria recursivamente.

```
// Quick Sort
void quickSort(int vet[], int inicio, int fim) {
    if (inicio < fim) {
        int pivo = separa(vet, inicio, fim);
        quickSort(vet, inicio, pivo);
        quickSort(vet, pivo + 1, fim);
    }
}
```

2.1 Resultados

A tabela a seguir apresenta o tempo de execução em segundos para cada algoritmo e tamanho de entrada:

ALGORITMO DE ORDENAÇÃO	TAMANHO DO VETOR		
	10^4	10^5	10^6
SELECTION SORT	0,049301	4,912422	452,488
INSERTION SORT	0,000357	0,004241	307,934121
MERGE SORT	0,001018	0,011615	0,111113
QUICK SORT	0,000656	0,006439	0,64949

A tabela comparativa dos tempos de execução para diferentes tamanhos de entrada revela diferenças significativas no desempenho entre os algoritmos. O Selection Sort e o Insertion Sort apresentaram tempos bastante elevados em relação ao Merge Sort e Quick Sort, chegando a mais de 12 horas, como foi o caso do Selection Sort, para ordenar um array de 10^6 elementos. Isso ocorre porque ambos têm complexidade quadrática O(n2), ou seja, o tempo cresce quadraticamente conforme o tamanho do input aumenta.

Já o Merge Sort e Quick Sort, por possuírem complexidade O(nlogn), tiveram desempenhos muito superiores. Para 10^6 elementos, o Merge Sort levou aproximadamente 0,111113 segundos e o Quick Sort pouco mais de 0,64949 segundos. Isso demonstra que a estratégia de divisão e conquista utilizada por esses algoritmos é bastante eficiente para ordenação de grandes volumes de dados.

Dentre os algoritmos de complexidade nlogn, o Quick Sort se mostrou ligeiramente superior ao Merge Sort. Isso pode ser explicado por otimizações como a escolha do pivô, que permite ao Quick Sort obter partições mais balanceadas a cada etapa.

Portanto, fica evidente que para ordenar arrays de tamanho considerável, algoritmos de ordenação mais eficientes como Quick Sort e Merge Sort são amplamente superiores e devem ser preferidos em relação aos métodos quadráticos como Insertion e Selection Sort. A diferença de performance se torna ainda mais crítica à medida que o tamanho dos dados de entrada cresce.

4. Conclusões

Os resultados dos testes indicam claramente que algoritmos de complexidade O(n log n), como o Merge Sort e o Quick Sort, superam significativamente os algoritmos de complexidade quadrática O(n^2), como o Selection Sort e o Insertion Sort, quando se trata de processar entradas grandes. Esse fenômeno é de extrema importância para a eficiência e o desempenho de algoritmos em muitos contextos computacionais.

A conclusão principal a ser tirada desses testes é que a escolha do algoritmo de classificação apropriado é fundamental para lidar com grandes conjuntos de dados. Quando a eficiência é uma preocupação, os algoritmos de complexidade O(n log n) são claramente superiores aos algoritmos de complexidade O(n^2). Isso pode resultar em economia significativa de tempo e recursos computacionais, especialmente em aplicações em que o processamento de grandes volumes de dados é uma tarefa frequente.

Dentro da categoria de algoritmos O(n log n), o Quick Sort se destacou como o melhor em termos de desempenho nos testes realizados. Isso pode ser atribuído às suas características intrínsecas, como a divisão eficiente do conjunto de dados e o uso de um pivô que minimiza o número de comparações necessárias para a classificação. Como resultado, o Quick Sort é uma escolha sólida quando a eficiência é uma prioridade.

No entanto, é importante notar que o desempenho de um algoritmo pode variar dependendo do tipo de dados de entrada e do ambiente de execução. Portanto, é essencial considerar o contexto específico ao escolher um algoritmo de classificação e, quando apropriado, realizar testes adicionais para confirmar sua eficácia em situações reais.

Em resumo, os testes experimentais destacam a importância de escolher algoritmos de classificação apropriados para lidar com grandes conjuntos de dados e demonstram que algoritmos de complexidade O(n log n), como o Quick Sort, são uma escolha sólida para otimizar o desempenho em cenários desse tipo.