**UNIVERSIDADE FEDERAL DE RORAIMA**

**CENTRO DE CIÊNCIA E TECNOLOGIA**

**DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO**

**DCC302 - ESTRUTURAS DE DADOS I - 2024.2**

**FILIPE DWAN PEREIRA**

**LEONARDO VINÍCIUS LIMA CASTRO**

**ARTHUR CORREIA DE OLIVEIRA RAMOS**

**ALGORITMOS DE ORDENAÇÃO**

**BOA VISTA, RR**

**2024**

**LEONARDO VINÍCIUS LIMA CASTRO**

**ARTHUR CORREIA DE OLIVEIRA RAMOS**

**ALGORITMOS DE ORDENAÇÃO**

Trabalho da disciplina de Estrutura de Dados I do ano de 2024 apresentado à Universidade Federal de Roraima do curso de Bacharelado em ciência da computação.

Docente: Filipe Dwan Pereira

**BOA VISTA, RR**

**2024**

**INTRODUÇÃO**

Os algoritmos de ordenação e busca são essenciais para organizar e encontrar informações de forma eficiente. Alguns métodos, como Bubble Sort e Insertion Sort, são mais simples, mas lentos para grandes conjuntos de dados. Outros, como Quick Sort e Merge Sort, são mais rápidos e usados em aplicações reais. Já os algoritmos de busca, como Busca Sequencial e Busca Binária, ajudam a encontrar dados de maneira eficiente, dependendo se a lista está ordenada ou não. Cada algoritmo tem suas vantagens e desvantagens, e a escolha do melhor depende da situação.

**ALGORITMOS**

**Selection Sort**

O Selection Sort é um algoritmo de ordenação por seleção que busca repetidamente o menor (ou maior) elemento do vetor e o coloca na posição correta. Ele divide o vetor em uma parte ordenada e outra não ordenada, selecionando o menor elemento e trocando-o com o primeiro elemento não ordenado.

Abaixo será apresentado o algoritmo em C:

void selectionSort(int v[], int n) {

int i, j, min, temp;

for (i = 0; i < n - 1; i++) {

min = i;

for (j = i + 1; j < n; j++) {

if (v[j] < v[min]) {

min = j;

}

}

temp = v[i];

v[i] = v[min];

v[min] = temp;

}

}

Vantagens:

* Simples de implementar.
* Não requer memória extra além do vetor original.
* Útil para pequenas listas.

Desvantagens:

* Desempenho O(n²) no pior e no melhor caso.
* Ineficiente para listas grandes.
* Pouco adaptável a entradas parcialmente ordenadas.

**Bubble Sort**

O **Bubble Sort** funciona comparando elementos adjacentes e trocando-os caso estejam na ordem errada. Esse processo é repetido até que a lista esteja completamente ordenada.

Abaixo será apresentado o algoritmo em C:

void bubbleSort (int v[], int n) {

int i, j, temp;

for (i = 0; i < n - 1; i++) {

for (j = 0; j < n - i - 1; j++) {

if (v[j] > v[j + 1]) {

temp = v[j];

v[j] = v[j + 1];

v[j + 1] = temp;

}

}

}

}

Vantagens:

* Simples e fácil de entender.
* Pode detectar listas já ordenadas rapidamente (com uma versão otimizada).

Desvantagens:

* Ineficiente para listas grandes, com O(n²) no pior e médio caso.
* Muitas trocas de elementos, tornando-o mais lento que Selection Sort para algumas entradas.

**Radix Sort**

O Radix Sort é um algoritmo de ordenação não comparativo que classifica os números por seus dígitos, utilizando contagem para ordenar em cada posição decimal. Ele é eficiente para ordenar inteiros e cadeias de caracteres de tamanho fixo.

Abaixo será apresentado o algoritmo em C:

#define MAX 10

void countingSort(int v[], int n, int exp) {

int output[n], count[MAX] = {0}, i;

for (i = 0; i < n; i++)

count[(v[i] / exp) % 10]++;

for (i = 1; i < MAX; i++)

count[i] += count[i - 1];

for (i = n - 1; i >= 0; i--) {

output[count[(v[i] / exp) % 10] - 1] = v[i];

count[(v[i] / exp) % 10]--;

}

for (i = 0; i < n; i++)

v[i] = output[i];

}

void radixSort(int v[], int n) {

int max = v[0], exp;

for (int i = 1; i < n; i++)

if (v[i] > max)

max = v[i];

for (exp = 1; max / exp > 0; exp \*= 10)

countingSort(v, n, exp);

}

Vantagens:

* Complexidade O(nk), onde *k* é o número de dígitos do maior número, o que o torna mais rápido que algoritmos quadráticos para certos casos.
* Ideal para ordenar grandes quantidades de inteiros.

Desvantagens:

* Não funciona bem para tipos de dados que não podem ser representados como números fixos.
* Requer memória extra para os buckets.
* Depende da base escolhida para os números.

**Merge Sort**

O Merge Sort é um algoritmo de ordenação baseado na estratégia Dividir para Conquistar, dividindo recursivamente a lista em partes menores até que cada parte tenha apenas um elemento, e então mesclando as partes ordenadas.

Abaixo será apresentado o algoritmo em C:

void merge(int v[], int esq, int meio, int dir) {

int i, j, k;

int n1 = meio - esq + 1;

int n2 = dir - meio;

int E[n1], D[n2];

for (i = 0; i < n1; i++)

E[i] = v[esq + i];

for (j = 0; j < n2; j++)

D[j] = v[meio + 1 + j];

i = 0;

j = 0;

k = esq;

while (i < n1 && j < n2) {

if (E[i] <= D[j]) {

v[k] = E[i];

i++;

} else {

v[k] = D[j];

j++;

}

k++;

}

while (i < n1) {

v[k] = E[i];

i++;

k++;

}

while (j < n2) {

v[k] = D[j];

j++;

k++;

}

}

void mergeSort(int v[], int esq, int dir) {

if (esq < dir) {

int meio = esq + (dir - esq) / 2;

mergeSort(v, esq, meio);

mergeSort(v, meio + 1, dir);

merge(v, esq, meio, dir);

}

}

Vantagens:

* Complexidade O(n log n) no pior, médio e melhor caso.
* Estável, ou seja, mantém a ordem relativa dos elementos iguais.
* Bom para ordenar grandes conjuntos de dados.

Desvantagens:

* Requer memória extra O(n) para armazenar os subarrays temporários.
* Pode ser mais lento que algoritmos simples para pequenas entradas devido à sobrecarga de chamadas recursivas.

**CONCLUSÃO**

O estudo desses algoritmos mostra que não existe uma única solução ideal, mas sim a mais adequada para cada caso. Métodos simples funcionam bem para listas pequenas, enquanto algoritmos mais avançados são melhores para grandes volumes de dados. A busca eficiente depende da organização dos dados, tornando a ordenação muitas vezes um passo necessário. Conhecer essas técnicas ajuda a melhorar o desempenho dos programas e a tomar melhores decisões na hora de processar informações.

**REFERÊNCIAS**

**CELES, Waldemar; CERQUEIRA, Renato; RANGEL, José Lucas. *Introdução à Estrutura de Dados*. 5. ed. Rio de Janeiro: Elsevier, 2016.**

**CORMEN, Thomas H.; LEISERSON, Charles E.; RIVEST, Ronald L.; STEIN, Clifford. *Introduction to Algorithms*. 3rd ed. Cambridge: MIT Press, 2009.**