**Relatório – Algoritmos de Ordenção - Estrutura de Dados I**

**Nome: Flávio Alves Wildner Real Rosa**

**Prof.: Leandro Alvim**

# Introdução

Algoritmos de ordenação são técnicas para colocar os elementos de uma lista(vetor) em ordem crescente ou decrescente. Para isso poderá ser utilizado diferentes algoritmos, cada um com uma certa vantagem em cada caso. As ordens mais usadas são a numérica e a lexicográfica.

O objetivo de se ordenar listas, é facilitar a busca por um elemento. Em uma busca binária por exemplo, é necessário que a lista esteja ordenada. Exibir dados ordenados para o usuário, como em um ranking é mais fácil analisar o desempenho dos candidatos com essa lista ordenada.

# Objetivo

O objetivo desse relatório é auxiliar o programador a escolher qual o melhor algoritmo de ordenção para seu tipo de aplicação a ser desenvolvida.

Neste relatório abordaremos as diferentes técnicas de ordenação de vetores, citando vantagens e desvantagens para cada caso com base na comparação entre os algoritmos. Serão utilizados os algoritmos: selection sort, counting sort, heap sort e radix sort.

# Descrição dos algoritmos

* **Selection sort:**

Percorre-se cada posição permutando com menor valor encontrado a partir dele. Complexidade O(n²).

Pseudocódigo:

INICIO

//declarar as variáveis:

Inteiro i, j, aux, menor, tamanho

Inteiro a[tamanho]

//ordenar o vetor:

Para i de 0 ate (tamanho – 1) //Percorre cada posição do vetor

menor <- i

Para j De i+1 Até 9 //Percorre todas as posições a partir da posição i

Se a[menor] > a[j] então //Se o valor for menor que a posição atual armazena-se na variável menor

menor <- j

Fimse

Fim Para

Se menor i != menor então //Se encontrado valor menor que a posição atual, permuta-se.

aux <- a[menor]

a[menor] <- a[i]

a[i] <- aux

Fim se

Fim para

* **Counting sort:**

A ideia é utilizar recipientes para organizar e classificar os dados e então retorna-los. Este tipo de algoritmo faz uso de um vetor auxiliar, onde é feito a separação e numeração das ocorrências dos dados de entrada, a qual os valores do vetor são usados como índices em um outro vetor.

1. Inicializar o vetor aux com o tamanho do maior valor do vetor com valores zero;
2. Percorre o vetor e cada valor encontrado ele incrementa 1 no índice correspondente no aux;
3. Percorre o vetor aux substituindo pela soma do valor da posição atual com o da poisção anterior.
4. Usa-se o valor do vetor principal como índice no vetor aux, obtendo o valor desse índice insere-se o valor na posição aux[índice – 1] do vetor resposta.

Pseudocódigo:

INICIO

//Variaveis

Inteiro i, tamanho

Inteiro aux[tamanho +1]

Para i de 0 ate (tamanho + 1) //Inicializa vetor aux com valores 0

aux[i] <- 0

Fim Para

Para i de 0 ate tamanho // cada valor encontrado no vetor ele incrementa 1 no índice correspondente no aux

aux[vetor[i]] += 1

Fim Para

//Usa-se o valor do vetor principal como índice no vetor aux, obtendo o valor desse índice insere-se o valor na posição aux[índice – 1] do vetor resposta.

Para i de 0 ate tamanho

Resultado[aux[vetor[i]--] <- vetor[i]

Fim Para

vetor <- resultado

Para i de 0 ate tamanho

Vetor[i] <- resultado[i]

Fim Para

* **Heap sort:**

O heapsort utiliza uma estrutura de dados chamada heap, para ordenar os elementos à medida que os insere na estrutura. Assim, ao final das inserções, os elementos podem ser sucessivamente removidos da raiz da heap, na ordem desejada, lembrando-se sempre de manter a propriedade de max-heap.

1. Monta-se uma Max-Heap com os valores do vetor;
2. Retira-se o elemento heap[0] e coloque na última posição do vetor, na heap[0] coloque o último elemento da heap.
3. Reorganize a Max-Heap.
4. Repita o passo 2 e 3 até que a Max-Heap seja vazia.

Pseudocódigo:

heapSort( int a[ ], int n )

INICIO

Para k = n/2 - 1 até 0

siftDown(k, a, n)

Para k = n-1 até 1

v = a[0] // Maior valor da Heap

a[0] = a[k]

siftDown( 0, a, k) // Tamanho da heap é k

a[k] = v //a[k] não está mais na heap

Fim

// Heap de a[0] até a[N-1], tamanho N

siftDown(int k, int a[ ], int N)

INICIO

v = a[k]

j = 2k + 1 // Filho esquerdo de k

Enquanto( j ≤ N-1 ) // Enquanto filho esquerdo está na heap

if( j < N-1 && a[j] < a[j+1])

++j

if( v >= a[j] )

Para

a[k] = a[j]

k = j

j =2k+1

a[k] = v

Fim

* **Radix sort:**

O Radix sort baseia-se na ordenação por dígito dos elementos, por exemplo, se o maior valor do elemento tem 6 dígitos então teremos 6 etapas, começando pelo dígito mais significativo ou menos significativo. Utilizamos outro algoritmo para ordenar o vetor, assumindo o dígito atual como chave. Esse algoritmo deverá ser estável.

Exemplo começando pelo menos significativo:

329 720 720 329

457 355 329 355

657 436 436 436

839 457 839 457

436 657 355 657

720 329 457 720

355 839 657 839

Pseudocódigo(usando Bucket sort como ordenação):

inteiro i;

inteiro b[tamanho];

inteiro maior <- vetor[0];

inteiro exp <- 1;

Para i=0 até tamanho

Se(vetor[i] > maior)

maior <- vetor[i]

Fim Para

//Ordena usando o bucket sort baseado no dígito atual.

Enquanto (maior / exp > 0)

inteiro bucket[10]

Para i=0 até tamanho

bucket[(vetor[i] / exp) % 10] += 1

Fim Para

Para i=0 até 10

bucket[i] += bucket[i - 1]

Fim Para

Para i=tamanho-1 até 0

b[--bucket[(vetor[i] / exp) % 10]] <- vetor[i]

Fim Para

Para i=0 até tamanho

vetor[i] <- b[i]

Fim Para

exp \*= 10

Fim Enquanto

# Metodologia das comparações

Diferentes algoritmos se comportam de formas diferentes quanto ao desempenho dependendo do vetor de entrada. As características do vetor são: número de elementos, faixa das chaves(chave máxima) e a forma como está organizado podendo estar aleatório ou semi-ordenado.

Por isso foram realizados dois testes, o primeiro variando o número de elementos com a faixa das chaves igual ao número de elementos para três formas de vetor: gerado aleatoriamente, ordenado em ordem crescente e decrescente. O segundo teste com um número fixo(15.000) de elementos mas variando a faixa de chaves.

As especificações do computador onde foram realizados os testes são: Processador Intel Core i5 4440 3.1 Ghz, 8GB de memória DDR3 1600MHz, Windows 10.

# Resultados

* **Selection Sort:**

Primeiro teste(diferentes tamanhos do vetor):

Segundo teste(diferentes variações de chave(0 –> n) para um vetor fixo de 15.000 elementos):

* **Counting sort:**
* **Radix sort:**
* **Heap sort:**

# Conclusões

De acordo com os testes, concluímos que para os algoritmos citados temos sua vantagens e desvantagens.

**Selection Sort:**

Vantagens: Muito rápido para listas pequenas com faixa grande de chaves;

Implementação simples;

Não necessita de vetor adicional.

Desvantagens: Extremamente ineficiente para listas grandes.

**Counting Sort:**

Vantagens: Muito rápido para listas grandes semi--ordenadas;

Não realiza comparações;

Desvantagens: Necessita de dois vetores adicionais;

Ineficiente para listas com faixa grande de chaves.

**Radix Sort:**

Vantagens: Muito rápido para grandes vetores com grande faixa de chaves com valores aleatórios;

Desvantagens: Ineficiente para listas semi-ordenadas.

**Heap Sort:**

Vantagens: Eficiente para vetores semi-ordenados com grande faixa de chaves.

Desvantagens: Ineficiente para listas com organização de chaves aleatórias;

Implementação complexa.