Trabalho Final de Estrutura de Dados I

1)

**Selection Sort:**

**Definição**

O selection sort é baseado em se passar o menor valor do vetor para a primeira posição (ou o maior dependendo da ordem desejada), depois o de segundo menor valor para a segunda posição, e assim é feito sucessivamente com os n − 1 elementos restantes, até os últimos dois que restarem.

**Algoritmo em C**

void selection\_sort(int num[], int tam) {

int i, j, min, aux;

for (i = 0; i < (tam-1); i++)

{

min = i;

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

if(num[j] < num[min])

min = j;

}

if (num[i] != num[min]) {

aux = num[i];

num[i] = num[min];

num[min] = aux;

}

}

}

**Vantagens**

* Um algoritmo simples de ser implementado em comparação aos demais.
* Não necessita de um vetor auxiliar (in-place).
* Por não usar um vetor auxiliar para realizar a ordenação, ele ocupa menos memória.
* Uns dos mais velozes na ordenação de vetores de tamanhos pequenos.

**Desvantagens**

* Um dos mais lentos para vetores de tamanhos grandes.
* Não é estável.
* Faz sempre n² comparações, independente do vetor está ordenado ou não.

https://pt.wikipedia.org/wiki/Selection\_sort

**Insertion Sort:**

**Definição**

É o algoritmo de ordenação que, dado uma estrutura (array, lista) constrói uma matriz final com um elemento de cada vez, uma inserção por vez. Assim como algoritmos de ordenação quadrática, é bastante eficiente para problemas com pequenas entradas, sendo o mais eficiente entre os algoritmos desta ordem de classificação.

**Algoritmo**

void insertionSort(int original[], int length) {

int i, j, atual;

for (i = 1; i < length; i++) {

atual = original[i];

for (j = i - 1; (j >= 0) && (atual < original[j]); j--) {

original[j + 1] = original[j];

}

original[j+1] = atual;

}

}

**Vantagens:**

* É o método a ser utilizado quando o arquivo está "quase" ordenado
* É um bom método quando se desejar adicionar poucos elementos em um arquivo já ordenado, pois seu custo é linear.
* É estável.

**Desvantagens:**

* Alto custo de movimentação de elementos no vetor.

https://pt.wikipedia.org/wiki/Insertion\_sort

**Bubble Sort:**

**Definição:**

É um algoritmo de ordenação dos mais simples. A ideia é percorrer o vetor diversas vezes, e a cada passagem fazer flutuar para o topo o maior elemento da sequência. Essa movimentação lembra a forma como as bolhas em um tanque de água procuram seu próprio nível, e disso vem o nome do algoritmo.

**Algoritmo**

**procedure** bubbleSort( A **:** lista de itens ordenaveis ) **defined as:**

**do**

trocado := false

**for each** i **in** 0 **to** length( A ) - 2 **do:**

// verificar se os elementos estão na ordem certa

**if** A[ i ] > A[ i + 1 ] **then**

// trocar elementos de lugar

trocar( A[ i ], A[ i + 1 ] )

trocado := true

**end if**

**end for**

// enquanto houver elementos sendo reordenados

**while** trocado

**end procedure**

**Vantagens:**

No melhor caso, o algoritmo executa {\displaystyle n} operações relevantes, onde {\displaystyle n} representa o número de elementos do vetor.

**Desvantagens:**

No pior caso, são feitas {\displaystyle n^{2}} operações. A complexidade desse algoritmo é de ordem quadrática. Por isso, ele não é recomendado para programas que precisem de velocidade e operem com quantidade elevada de dados.

https://pt.wikipedia.org/wiki/Bubble\_sort

**Radix Sort:**

**Definição:**

É um algoritmo de ordenação rápido e estável que pode ser usado para ordenar itens que estão identificados por chaves únicas. Cada chave é uma cadeia de caracteres ou número, e o *radix sort* ordena estas chaves em qualquer ordem relacionada com a lexicografia.

**Algoritmo em C**

void radixsort(int vetor[], int tamanho) {

int i;

int \*b;

int maior = vetor[0];

int exp = 1;

b = (int \*)calloc(tamanho, sizeof(int));

for (i = 0; i < tamanho; i++) {

if (vetor[i] > maior)

maior = vetor[i];

}

while (maior/exp > 0) {

int bucket[10] = { 0 };

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

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

for (i = 1; i < 10; i++)

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

for (i = tamanho - 1; i >= 0; i--)

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

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

vetor[i] = b[i];

exp \*= 10;

}

free(b);

}

**Vantagens:**

* **N**ão realiza comparações;
* É estável, preserva a ordem de chaves iguais;
* Tempo Linear.

**Desvantagens:**

* Precisa de mais memória para ordenar o conjunto;
* O algoritmo é se torna muito caro quando o vetor chave é muito extenso.

<https://www.google.com.br/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&cad=rja&uact=8&ved=0ahUKEwiO5O3e0trYAhWGC5AKHV6-ABwQFggpMAA&url=http%3A%2F%2Fwww.joinville.udesc.br%2Fportal%2Fprofessores%2Fkariston%2Fmateriais%2FSlides_Exercicios_Radix_Bucket_CountingSort_v1.pptx&usg=AOvVaw0cNqFoKhuDUgpkYveiWw5_>

**Quick Sort:**

**Definição**

Funciona ao separar grupos dentro do vetor, e ordenar esses grupos, logo conforme vai avançando os grupos vão ficando ordenados.

**Algoritmo**

#include <stdio.h>  
   
void quicksort(int \*A, int len);  
   
int main (void) {  
 int a[] = {4, 65, 2, -31, 0, 99, 2, 83, 782, 1};  
 int n = sizeof a / sizeof a[0];  
   
 int i;  
 for (i = 0; i < n; i++) {  
 [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("%d ", a[i]);  
 }  
 [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("\n");  
   
 quicksort(a, n);  
   
 for (i = 0; i < n; i++) {  
 [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("%d ", a[i]);  
 }  
 [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("\n");  
   
 return 0;  
}  
   
void quicksort(int \*A, int len) {  
 if (len < 2) return;  
   
 int pivot = A[len / 2];  
   
 int i, j;  
 for (i = 0, j = len - 1; ; i++, j--) {  
 while (A[i] < pivot) i++;  
 while (A[j] > pivot) j--;  
   
 if (i >= j) break;  
   
 int temp = A[i];  
 A[i] = A[j];  
 A[j] = temp;  
 }  
   
 quicksort(A, i);  
 quicksort(A + i, len - i);  
}

**Vantagens:**

**Desvantagens:**

<https://terminaldeinformacao.com/2015/06/15/shell-sort-e-quick-sort/>

http://rosettacode.org/wiki/Sorting\_algorithms/Quicksort#C

**Merge Sort:**

**Definição:**

Sua ideia básica consiste em Dividir (o problema em vários subproblemas e resolver esses subproblemas através da recursividade) e conquistar (após todos os subproblemas terem sido resolvidos ocorre a conquista que é a união das resoluções dos subproblemas). Como o algoritmo *Merge Sort* usa a recursividade, há um alto consumo de memória e tempo de execução, tornando esta técnica não muito eficiente em alguns problemas.

**Algoritmo:**

void merge(int vetor[], int comeco, int meio, int fim) {

int com1 = comeco, com2 = meio+1, comAux = 0, tam = fim-comeco+1;

int \*vetAux;

vetAux = (int\*)malloc(tam \* sizeof(int));

while(com1<=meio && com2<=fim){

if(vetor[com1] <= vetor[com2]){

vetAux[comAux] = vetor[com1];

com1++;

}else{

vetAux[comAux] = vetor[com2];

com2++;

}

comAux++;

}

while(com1<=meio){ //Caso ainda haja elementos na primeira metade

vetAux[comAux] = vetor[com1];

comAux++;com1++;

}

while(com2<=fim){ //Caso ainda haja elementos na segunda metade

vetAux[comAux] = vetor[com2];

comAux++;com2++;

}

for(comAux=comeco;comAux<=fim;comAux++){ //Move os elementos de volta para o vetor original

vetor[comAux] = vetAux[comAux-comeco];

}

free(vetAux);

}

void mergeSort(int vetor[], int comeco, int fim){

if (comeco < fim) {

int meio = (fim+comeco)/2;

mergeSort(vetor, comeco, meio);

mergeSort(vetor, meio+1, fim);

merge(vetor, comeco, meio, fim);

}

}

**Vantagens:**

**Desvantagens:**

* Utiliza funções recursivas;
* Gasto extra de memória. O algoritmo cria uma cópia do vetor para cada nível da chamada recursiva, totalizando um uso adicional de memória igual a (n log n).

https://pt.wikipedia.org/wiki/Merge\_sort

**Busca sequencial:**

**Definição:**

**Algoritmo:**

**Vantagens:**

**Desvantagens:**

**Busca Binária:**

**Definição:**

É um algoritmo de busca em vetores que segue o paradigma de divisão e conquista. Ela parte do pressuposto de que o vetor está ordenado e realiza sucessivas divisões do espaço de busca comparando o elemento buscado (chave) com o elemento no meio do vetor. Se o elemento do meio do vetor for a chave, a busca termina com sucesso. Caso contrário, se o elemento do meio vier antes do elemento buscado, então a busca continua na metade posterior do vetor. E finalmente, se o elemento do meio vier depois da chave, a busca continua na metade anterior do vetor.

**Algoritmo:**

//Implementação Iterativa:

int PesquisaBinaria (int vet[], int chave, int Tam)

{

int inf = 0; // limite inferior (o primeiro índice de vetor em C é zero )

int sup = Tam-1; // limite superior (termina em um número a menos. 0 a 9 são 10 números)

int meio;

while (inf <= sup)

{

meio = (inf + sup)/2;

if (chave == vet[meio])

return meio;

if (chave < vet[meio])

sup = meio-1;

else

inf = meio+1;

}

return -1; // não encontrado

}

//Implementação Recursiva:

// x => chave | v[] => vetor ordenado | e => limite inferior (esquerda) | d => limite superior (direita)

int PesquisaBinaria (int x, int v[], int e, int d)

{

int meio = (e + d)/2;

if (v[meio] == x)

return meio;

if (e >= d)

return -1; // não encontrado

else

if (v[meio] < x)

return PesquisaBinaria(x, v, meio+1, d);

else

return PesquisaBinaria(x, v, e, meio-1);

}

**Vantagens:**

**Desvantagens:**

https://pt.wikipedia.org/wiki/Pesquisa\_bin%C3%A1ria