**JORGE SIQUEIRA SERRÃO**

**WILLIAM THIAGO**

**ORDENAÇÃO E BUSCA**

**Selection Sort**

A ordenação por seleção (do inglês, selection sort) é um algoritmo de ordenação baseado em se passar sempre o menor valor do vetor para a primeira posição (ou o maior dependendo da ordem requerida), 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 elementos.

* Vantagens
* Ele é 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.
* Ele é uns dos mais velozes na ordenação de vetores de tamanhos pequenos.
* Desvantagens
* Ele é um dos mais lentos para vetores de tamanhos grandes.
* Ele não é estável.
* Ele faz sempre {\displaystyle (n^{2}-n)/2}(n²-n)/2 comparações, independente do vetor está ordenado ou não.
* Código:

void selection\_sort(int \*a,int tam)

{

printf("\n===SELECTION SORT===\n");

int i, j, k, tmp, troca;

int qtroca=0;

int qcomparacao=0;

clock\_t tic = clock();

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

{

troca = 0;

k = i;

tmp = a[i];

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

{

if(a[j] < tmp)

{

k = j;

tmp = a[j];

troca = 1;

qcomparacao++;

}

}

if(troca)

{

a[k] = a[i];

a[i] = tmp;

qtroca++;

}

}

clock\_t toc = clock();

printf("TROCAS: %d\n",qtroca);

printf("COMPARACOES: %d\n",qcomparacao);

printf("TEMPO: %f milisegundos\n", (double)(toc - tic) / CLOCKS\_PER\_SEC);

}

**Insertion Sort**

Insertion Sort, ou ordenação por inserção, 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.

* 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.
* O algoritmo de ordenação por inserção é estável.
* Desvantagens
* Alto custo de movimentação de elementos no vetor.

Código:

void insertionSort(int arr[], int n)

{

int qtroca=0;

int qcomparacao=0;

clock\_t tic = clock();

printf("\n===INSERTION SORT===\n");

int i, key, j;

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

qtroca++;

key = arr[i];

j = i - 1;

while (j >= 0 && arr[j] > key) {

qcomparacao++;

arr[j + 1] = arr[j];

j = j - 1;

}

arr[j + 1] = key;

}

clock\_t toc = clock();

printf("TROCAS: %d\n",qtroca);

printf("COMPARACOES: %d\n",qcomparacao);

printf("TEMPO: %f milisegundos\n", (double)(toc - tic) / CLOCKS\_PER\_SEC);

}

**Bubble Sort**

O bubble sort, ou ordenação por flutuação (literalmente "por bolha"), é um algoritmo de ordenação dos mais simples. A ideia é percorrer o vector 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.

* Vantagens
* Simplicidade do algoritmo
* Estável
* Desvantagens
* Lentidão
* Código:

void bubble\_sort (int vetor[], int n) {

int qtroca=0;

int qcomparacao=0;

clock\_t tic = clock();

printf("\n===BUBBLE SORT===\n");

int k, j, aux;

for (k = 1; k < n; k++) {

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

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

aux = vetor[j];

vetor[j] = vetor[j + 1];

vetor[j + 1] = aux;

qtroca++;

qcomparacao++;

}

}

}

clock\_t toc = clock();

printf("TROCAS: %d\n",qtroca);

printf("COMPARACOES: %d\n",qcomparacao);

printf("TEMPO: %f milisegundos\n", (double)(toc - tic) / CLOCKS\_PER\_SEC);

}

**Radix Sort**

O Radix sort 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.

* Vantagens
* Estável
* Não compara as chaves
* Desvantagens
* Nem sempre é fácil otimizar a inspeção de dígitos
* Depende do hardware
* Só é bom se o número de dígitos for pequeno
* Em geral o número de digítos tem crescimento O(lg(n))
* Código:

int radixtroca=0;

int radixcomp=0;

int getMax(int array[], int n)

{

int max = array[0];

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

if (array[i] > max)

max = array[i];

radixcomp+=1;

return max;

}

void countingSort(int array[], int size, int place)

{

int output[size + 1];

int max = (array[0] / place) % 10;

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

{

if (((array[i] / place) % 10) > max)

max = array[i];

radixcomp++;

}

int count[max + 1];

for (int i = 0; i < max; ++i)

count[i] = 0;

radixtroca++;

for (int i = 0; i < size; i++)

count[(array[i] / place) % 10]++;

radixtroca++;

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

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

radixtroca++;

for (int i = size - 1; i >= 0; i--)

{

output[count[(array[i] / place) % 10] - 1] = array[i];

count[(array[i] / place) % 10]--;

radixtroca++;

}

for (int i = 0; i < size; i++)

array[i] = output[i];

radixcomp++;

radixtroca++;

}

void radixsort(int array[], int size)

{

int max = getMax(array, size);

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

countingSort(array, size, place);

}

**Quick Sort**

O algoritmo quicksort é um método de ordenação muito rápido e eficiente, inventado por C.A.R. Hoare em 1960, quando visitou a Universidade de Moscovo como estudante. Naquela época, Hoare trabalhou em um projeto de tradução de máquina para o National Physical Laboratory. Ele criou o quicksort ao tentar traduzir um dicionário de inglês para russo, ordenando as palavras, tendo como objetivo reduzir o problema original em subproblemas que possam ser resolvidos mais fácil e rápido. Foi publicado em 1962 após uma série de refinamentos. O quicksort é um algoritmo de ordenação por comparação não-estável.

* Vantagens
* Melhor opção para ordenar vetores grandes
* Muito rápido por que o laço interno é simples
* Memória auxiliar para a pilha de recursão é pequena
* Complexidade no caso médio é O (n lg( n))
* Desvantagens
* Não é estável (não conhecemos forma eficiente para tornar o quicksort estável)
* Pior caso é quadrático
* Código:

int quickcomp = 0;

int quicktroca = 0;

void quicksort(int \*number,int first,int last){

int i, j, pivot, temp;

if(first<last){

pivot=first;

i=first;

j=last;

quickcomp++;

quicktroca+=2;

while(i<j){

quickcomp++;

while(number[i]<=number[pivot]&&i<last){

quickcomp++;

i++;}

while(number[j]>number[pivot]){

j--;

quickcomp++;}

if(i<j){

temp=number[i];

number[i]=number[j];

number[j]=temp;

quickcomp++;

quicktroca+=2;

}

}

temp=number[pivot];

number[pivot]=number[j];

number[j]=temp;

quicktroca+=2;

quicksort(number,first,j-1);

quicksort(number,j+1,last);

}

}

**Merge Sort**

O merge sort, ou ordenação por mistura, é um exemplo de algoritmo de ordenação por comparação do tipo dividir-para-conquistar. 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.

* Vantagens
* É um algorito de ordenação de fácil implementação
* Útil para aplicação com restrição de tempo
* Passível de ser transformando em estável
* O Merge Sort é O(n lon n)
* Desvantagens
* Alto consumo de memória, devido à série de chamdas recursivas
* Utililiza memória auxiliar – O(n).
* Código:

int mergecomp = 0;

int mergetroca = 0;

void merge(int arr[], int l, int m, int r)

{

int i, j, k;

int n1 = m - l + 1;

int n2 = r - m;

/\* create temp arrays \*/

int L[n1], R[n2];

/\* Copy data to temp arrays L[] and R[] \*/

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

L[i] = arr[l + i];

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

R[j] = arr[m + 1+ j];

i = 0;

j = 0;

k = l;

while (i < n1 && j < n2)

{

mergecomp++;

if (L[i] <= R[j])

{

arr[k] = L[i];

i++;

mergetroca++;

mergecomp++;

}

else

{

arr[k] = R[j];

j++;

mergetroca++;

mergecomp++;

}

k++;

}

while (i < n1)

{

arr[k] = L[i];

i++;

k++;

mergetroca++;

mergecomp++;

}

while (j < n2)

{

arr[k] = R[j];

j++;

k++;

mergetroca++;

mergecomp++;

}

}

void mergeSort(int \*arr, int l, int r)

{

if (l < r)

{

mergecomp++;

int m = l+(r-l)/2;

mergeSort(arr, l, m);

mergeSort(arr, m+1, r);

merge(arr, l, m, r);

}

}

**Heap Sort**

Tem um desempenho em tempo de execução muito bom em conjuntos ordenados aleatoriamente, tem um uso de memória bem comportado e o seu desempenho em pior cenário é praticamente igual ao desempenho em cenário médio. Alguns algoritmos de ordenação rápidos têm desempenhos espectacularmente ruins no pior cenário, quer em tempo de execução, quer no uso da memória. O heapsort trabalha no lugar e o tempo de execução em pior cenário para ordenar n elementos é de O (n lg n). Lê-se logaritmo (ou log) de "n" na base 2. Para valores de n, razoavelmente grandes, o termo log n é quase constante, de modo que o tempo de ordenação é quase linear com o número de itens a ordenar.

* Vantagens
* Comportamento O(n lg n) no pior caso
* Desvantagens
* Não é estável
* Não é tão rápido quanto o quicksort por que o laço interno (refaz\_cima\_baixo) realiza mais operações que o particionamento do quicksort
* Código:

int heapcomp=0;

int heaptroca=0;

void heapsort(int \*vet, int n) {

int i, tmp;

for (i = (n / 2); i >= 0; i--) {

peneira(vet, i, n - 1);

heapcomp++;

}

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

heapcomp++;

tmp = vet[0];

vet[0] = vet[i];

vet[i] = tmp;

heaptroca++;

peneira(vet, 0, i-1);

}

}

void peneira(int \*vet, int raiz, int fundo) {

int pronto, filhoMax, tmp;

pronto = 0;

while ((raiz\*2 <= fundo) && (!pronto)) {

if (raiz\*2 == fundo) {

filhoMax = raiz \* 2;

heapcomp++;

}

else if (vet[raiz \* 2] > vet[raiz \* 2 + 1]) {

filhoMax = raiz \* 2;

heaptroca++;

}

else {

filhoMax = raiz \* 2 + 1;

heaptroca++;

}

if (vet[raiz] < vet[filhoMax]) {

heapcomp++;

tmp = vet[raiz];

vet[raiz] = vet[filhoMax];

vet[filhoMax] = tmp;

raiz = filhoMax;

heaptroca++;

}

else {

pronto = 1;

}

}

}

**Busca sequencial**

A busca sequencial é a técnica mais simples de realizar uma busca em uma lista de dados desordenados. Ela visa procurar o valor através de comparações sucessivas a partir do primeiro elemento (ou último) até que se encontre o valor desejado ou até que os elementos da estrutura se esgotem. Pode-se utilizar vetor, lista encadeada ou arquivo binário como estrutura de dados.

* Vantagem
* Os itens na tabela poderão ser examinados seqüencialmente sem que todos os registros precisem ser acessados
* O tempo de busca diminui consideravelmente
* Desvantagens
* A tabela tem que estar ordenada
* Exige espaço adicional para armazenar a(s) tabela(s) de índices
* Código:

int sequencial(int valor,int \*vetor){

for(int i=0;i<10;i++)

{

if (vetor[i]==valor)

{

return (i);

}

}

return(-1);

}

**Busca Binária**

A busca binária é um eficiente algoritmo para encontrar um item em uma lista ordenada de itens. Ela funciona dividindo repetidamente pela metade a porção da lista que deve conter o item, até reduzir as localizações possíveis a apenas uma.

* Vantagens
* Eficiência da busca
* Simplicidade da implementação
* Desvantagens
* Nem todo arranjo está ordenado
* Exige o uso de um arranjo para armazenar os dados
* Faz uso do fato de que os índices do vetor são inteiros consecutivos
* Inserção e remoção de elementos são ineficientes
* Realocação de elementos
* Código:

int PesquisaBinaria( int \* k, int chave , int N)

{

int inf,sup,meio;

inf=0;

sup=N-1;

while (inf<=sup)

{

meio=(inf+sup)/2;

if (chave==k[meio])

return meio;

else if (chave<k[meio])

sup=meio-1;

else

inf=meio+1;

}

return -1; /\* não encontrado \*/

}

**REFERÊNCIAS**

Wikipédia, a enciclopédia livre. **Selection sort**. Disponível em: <<https://pt.wikipedia.org/wiki/Selection_sort>>. Acesso em Nov de 2019.

Wikipédia, a enciclopédia livre. **Insertion sort**. Disponível em: <<https://pt.wikipedia.org/wiki/Insertion_sort>>. Acesso em Nov de 2019.

Wikipédia, a enciclopédia livre. **Radix sort**. Disponível em: <<https://pt.wikipedia.org/wiki/Radix_sort>>. Acesso em Nov de 2019.

Wikipédia, a enciclopédia livre. **Quicksort**. Disponível em: <<https://pt.wikipedia.org/wiki/Quicksort>>. Acesso em Nov de 2019.

Wikipédia, a enciclopédia livre. **Merge sort**. Disponível em: <<https://pt.wikipedia.org/wiki/Merge_sort>>. Acesso em Nov de 2019.

Wikipédia, a enciclopédia livre. **Heapsort**. Disponível em: <<https://pt.wikipedia.org/wiki/Heapsort>>. Acesso em Nov de 2019.

FILITTO. **Método de Busca Sequencial**. Disponível em: <<https://dfilitto.com.br/c/metodo-de-busca-sequencial/>>. Acesso em Nov de 2019.

Khan Academy. **Busca binária**. Disponível em: <<https://pt.khanacademy.org/computing/computer-science/algorithms/binary-search/a/binary-search>>. Acesso em Nov de 2019.

Souza, Jairo. **Método BubbleSort**. Disponível em: <<http://www.ufjf.br/jairo_souza/files/2009/12/2-Ordena%C3%A7%C3%A3o-BubbleSort.pdf>>. Acesso em Nov de 2019.

**Ordenação: Radixsort.** Disponível em: <<https://homepages.dcc.ufmg.br/~cunha/teaching/20121/aeds2/radixsort.pdf>>. Acesso em Nov de 2019.

**Quicksort**. Disponível em: <<https://homepages.dcc.ufmg.br/~cunha/teaching/20121/aeds2/quicksort.pdf>>. Acesso em Nov de 2019.

Andrade, Wárlley. **Merge Sort, Notas de estudo de Informática.** Disponível em: <<https://www.docsity.com/pt/merge-sort-8/4839179/>>. Acesso em Nov de 2019.

**Ordenação: Heapsort**. Disponível em: <<https://homepages.dcc.ufmg.br/~cunha/teaching/20121/aeds2/heapsort.pdf>>. Acesso em Nov de 2019.

**Métodos de busca**. Disponível em: <<http://wiki.icmc.usp.br/images/7/7c/M%C3%A9todos_de_busca_-_parte1_2010.pdf>>. Acesso em Nov de 2019.