**Selection Sort**

Selection sort é um algoritmo de ordenação que opera em uma complexidade C(n) = O(n²) , pois o número de operações sempre é o mesmo no melhor caso (em que os valores já estão ordenados) e no pior caso (em que os valores estão na ordem contrária da requerida) e é feito sempre n² vezes em relação ao tamanho do array.

O algoritmo funciona sempre pegando o menor valor da lista e colocando ele no começo da lista e depois o segundo menor e colocando ele na segunda posição e assim até o último elemento(isso contando com que o programa deseja ordenar em ordem crescente, já que se fosse em ordem decrescente seria o maior valor trabalhado no lugar dor menor).

Possui como uma das principais vantagens o fato de ser implementado de uma forma mais simples em relação aos demais algoritmos de comparação, não necessita de um vetor auxiliar o que torna melhor para aplicações que não podem usar muita memória, e é também um dos mais velozes na ordenação de vetores pequenos. Mas possui desvantagens também, que são o fato de que sempre faz n² comparações não importa se o vetor está ordenado ou não, é lento para vetores que possuem muitos elementos e dependendo do algoritmo não é estável.

Exemplo de algoritmo Selection Sort (em C):

void selection\_sort (int vetor[],int max) {

  int i, j, min, aux;

  for (i = 0; i < (max - 1); i++) {

    min = i;

    for (j = i+1; j < max; j++) {

      if (vetor[j] < vetor[min]) {

    min = j;

      }

    }

    if (i != min) {

      aux = vetor[i];

      vetor[i] = vetor[min];

      vetor[min] = aux;

    }

}

}

**Insertion Sort**

É um algoritmo de ordenação que assim como o selection sort é menos eficiente que algoritmos mais avançados, como por exemplo quicksort e o merge sort, para vetores de grande escala. E possui uma complexidade de O(n²) (assim como o Selection sort) mas apenas para os “casos médios” e “piores casos” já que se for o pior caso (ou seja ordem inversa da desejada ) e o “caso médio” (em que a lista está bagunçada) fazem com que cada iteração do looping passe por toda a parte do vetor e desloque ele, já no melhor caso em que o vetor já está ordenado ele fará uma execução linear, ou seja O(n).

O algoritmo de insertion sort começa analisando já o segundo valor do array com o primeiro e então, depois o terceiro com os dois anteriores, depois o quarto com os três anteriores e assim sucessivamente até o ultimo elemento do vetor. O código faz isso para analisar onde o valor será inserido, se será na ponta, no meio, ou se o valor fica na mesma posição.

Este algoritmo, apesar de possuir um alto custo computacional na movimentação dos elementos do vetor e não ser muito efetivo em vetores de grande tamanhos é um método bastante usado para coisas em que já está ‘quase’ ordenado e se for para adicionar poucos elementos em um arquivo já ordenado é uma boa opção também por ter um custo linear nesse caso.

Exemplo de algoritmo Insertion Sort (em C):

void insertionSort(int arr[], int n){

int i, j, valor;

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

valor = arr[i];

j = i – 1;

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

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

j = j – 1;

}

arr[j+1] = valor;

}

}

Bubble Sort

É um algoritmo de ordenação em que consiste percorrer o vetor várias vezes e então fazer em cada uma delas com que os valores maiores, se no caso o foco for organizar em ordem crescente o array, o valor ‘flutue’, literalmente como se fosse uma bolha, até a extremidade possível do vetor. E assim como os algoritmos supracitados também possui complexidade O(n²), mas no melhor caso isso se torna linear, ou seja, O(n).

E assim como o insertion sort e selection sort, ele possui a desvantagem de que é mais funcional em vetores menores, pois, a sua ordem ser quadrática(O[n²]) para a execução dessa ordenação faz com que não seja tão efetivo quanto outros já existentes para grandes vetores.

Mas possui uma boa vantagem, de ser um código de fácil compreensão, adaptação e fácil de se conseguir usar.

Considerando que a ordenação é para ordem crescente, o código consiste em percorrer o vetor, e sempre verificar se o valor que “está na vez” é maior que o próximo valor e se isso for verdade uma variável auxiliar recebe o valor que está sendo ‘analisado’ e na posição do valor que está sendo ‘analisado’ o próximo fica no lugar, e no lugar do próximo a partir do valor salvo na variável auxiliar ele é colocado nessa posição. E isso se repete N vezes (sendo N o tamanho do vetor)

Exemplo de algoritmo Bubble Sort (em C):

void BubbleSort(int numeros[], int TAM){

int contador, aux, i;

for (contador = 1; contador < TAM; contador++) {

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

if (numeros[i] > numeros[i + 1]) {

aux = numeros[i];

numeros[i] = numeros[i + 1];

numeros[i + 1] = aux;

}

}

}

}

**Radix Sort**

Este algoritmo é um pouco mais complexo de se compreender comparado aos anteriores, selection sort, inserction sort e bubble sort. Possui uma complexidade de tempo de O(nk) e complexidade de espaço de O(n+s) (sendo N o número de elementos, K tamanho da string e S tamanho do alfabeto), mas se o número de dígitos for pequeno ou constante o Radix Sort tem custo linear (O[n]).

Possui a vantagem de ser um algoritmo estável e não comparar chaves, como é feito nos outros supracitados, já que ele vai comparando ‘digito por digito’ para fazer a sua ordenação (seja ela em ordem crescente ou decrescente), é um dos algoritmos de ordenação mais rápidos e é possível fazer diferentes implementações do Radix sort dependendo do tipo de dado a ser trabalhado. Possui a desvantagem de que é melhor ser usado se a quantidade de dígitos do valor for pequena, então com dígitos de grandes escalas e em vetores muito grandes pode não ser tão interessante o seu uso, e nem sempre é de fácil otimização a inspeção desses dígitos, dependendo do hardware.

O algoritmo verifica digito por digito dos valores no vetor, considerando o objetivo de ordenar em ordem crescente, o algoritmo verifica qual o menor valor do digito menos significativo, ou seja, em [180,132,144,128,012] ele verificará que o menor valor dos dígitos menos significativos nesse vetor é o 0, o próximo o 2, depois o 4, depois o 8 e com isso ele faz o primeiro passo e deixa o vetor assim[180,132,012,144,128], depois verifica o digito do meio dos valores e com base neles faz outro passo e após verificar que o menor é 1, depois 2, depois 3, depois 4 deixa o vetor assim [012,128,132,144,180] e sem precisar verificar o digito mais significativo o vetor já está ordenado.

Exemplo de algoritmo Radix Sort (em C):

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

int i, \*b,exp = 1;

int maior = vetor[0];

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, 0, 0, 0, 0, 0, 0, 0, 0, 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);

}

Quick Sort

Este algoritmo de ordenação foi inventado por Charles Antony Richard Hoare, um cientista da computação britânico, que inclusive é ganhador do prêmio Turing (de 1980). É o algoritmo de ordenação mais utilizado no mundo.

Possui uma complexidade de O (N log N) para os melhores casos e casos médios, já o pior caso possui complexidade de O(n2). É um algoritmo extremamente utilizado pela sua grande eficácia em muitos casos. O Quick Sort é uma forma de árvore binária ordenada, em que ele organiza os valores corretamente na ordem, de forma implícita, e faz chamadas recursivas dela mesma para isso.

Uma da principais vantagens deste método é a sua velocidade que necessita apenas de O (N log N), capacidade de ordenar com facilidade vetores grandes, comparações em média para ordenar o vetor requerido, além de necessitar apenas de uma pequena pilha como memória auxiliar, mas também possui suas desvantagens como a sua implementação que é um pouco difícil já que um pequeno erro fará com que a recursividade não funcione e tamém não é um método estável(mas existe versão dele que consegue obter estabilidade mas com um custo extra).

O QuickSort usa o particionamento para efetuar o seu processo (dividir para conquistar), primeiro ele escolhe um elemento para servir de referência (chamado de pivô) e então ele reordena o vetor fazendo comparações com o valor do pivô. Os valores menores ou iguais ao pivô ficaram à esquerda do pivô e os maiores ou iguais ficaram à direita. E efetuando esses passos recursivamente até que o vetor esteja totalmente ordenado.

Exemplo de algoritmo Quick Sort (em C):

int partition( int \*vetorDesordenado, int posicaoInicio, int posicaoFim ){

int x,i,j,auxiliar;

x = vetorDesordenado[posicaoInicio];

i = posicaoInicio - 1;

j = posicaoFim + 1;

for(;;)

{

do { j--; } while( vetorDesordenado[j] > x );

do { i++; } while( vetorDesordenado[i] < x );

if (i < j)

{

auxiliar = vetorDesordenado[i];

vetorDesordenado[i] = vetorDesordenado[j];

vetorDesordenado[j] = auxiliar;

}

else

{

return j;

}

}

}

void quickSort( int \*vetorDesordenado, int posicaoInicio, int posicaoFim )

{

int pivot;

if (posicaoInicio < posicaoFim)

{

pivot = partition( vetorDesordenado, posicaoInicio, posicaoFim);

quickSort( vetorDesordenado, posicaoInicio, pivot);

quickSort(vetorDesordenado, pivot+1, posicaoFim);

}

}

**Merge Sort (ou ordenação por mistura)**

Este algoritmo, assim como o Quick Sort, é um algoritmo de comparação ‘dividir para conquistar’, criado por John Von Neumann, considerado um dos mais importantes matemáticos do século XX e um dos construtores do ENIAC. Consiste na ideia de criar uma sequência ordenada a partir de outras duas que também estejam ordenadas e para fazer isso ele divide o original em vários pares de dados. Possui complexidade O (n log n) para todos os casos, seja o caso médio, melhor ou pior, mas as vezes pode acontecer O(n), ser linear, no melhor caso.

Possui a desvantagem de precisar de muita memória já que ele requer o dobro, pois irá precisar de um vetor com as mesmas dimensões do vetor que está ordenado, além de utilizar funções recursivas. Mas é um algoritmo que é, na maioria do casos, estável além de ser de fácil implementação e uma complexidade de O (n log n) que permite com que ele seja um dos algoritmos de ordenação mais rápido.

O Merge Sort primeiro divide o algoritmo em duas grandes partes e então continua essa divisão de dados até que se tenha pares de dados e ao chegar nesses pares ele organiza quem é maior e quem é menor e então vai juntando com os outros pares e comparando, e assim organizando todos os dados na ordem desejada, chamando a própria função do Merge Sort(recursividade).

Exemplo de algoritmo Merge Sort (em C):

void mergeSort( int \*vetorDesorndeado, int posicaoInicio, int posicaoFim ) {

int i,j,k,metadeTamanho,\*vetorTemp;

if ( posicaoInicio == posicaoFim ) return;

metadeTamanho = ( posicaoInicio+posicaoFim )/2;

mergeSort( vetorDesorndeado, posicaoInicio, metadeTamanho);

mergeSort( vetorDesorndeado, metadeTamanho+1,posicaoFim );

i = posicaoInicio;

j = metadeTamanho+1;

k = 0;

vetorTemp = (int \*) malloc(sizeof(int) \* (posicaoFim-posicaoInicio+1));

while( i < metadeTamanho+1 || j < posicaoFim+1 ){

if ( i == metadeTamanho+1 ){

vetorTemp[k] = vetorDesorndeado[j];

j++;

k++;

}

else{

if (j==posicaoFim+1) {

vetorTemp[k] = vetorDesorndeado[i];

i++;

k++;

}

else {

if (vetorDesorndeado[i] < vetorDesorndeado[j]) {

vetorTemp[k] = vetorDesorndeado[i];

i++;

k++;

}

else{

vetorTemp[k] = vetorDesorndeado[j];

j++;

k++;

}

}

}

}

for( i = posicaoInicio; i <= posicaoFim; i++ ){

vetorDesorndeado[i] = vetorTemp[i-posicaoInicio];

}

free(vetorTemp);

}

**Busca binária**

É um algoritmo de busca que também segue o paradigma de ‘dividir para conquistar’ e possui a complexidade de O(log n) para o caso médio e pior caso, já no melhor caso a complexidade torna-se O(1).

A busca binária é feita para vetores em que os dados já estão ordenados, já que seu algoritmo já parte deste pressuposto. A principal vantagem deste algoritmo é a sua rapidez em achar o valor já que o vetor já está ordenado e assim o algoritmo consegue efetuar bem o seu propósito já que ela efetua poucos acessos no vetor para achar a posição do valor, mas possui a desvantagem de não conseguir efetuar a buscar em um vetor que não esteja devidamente ordenado.

O algoritmo, por já partir do pressuposto que o vetor está ordenado, ele primeiro vai ao valor que está no meio do vetor, se o valor já está lá a busca acaba ali mesmo e retorna o índice da posição do valor, mas se não achar imediatamente se o elemento que se encontra no meio vier antes do elemento que está sendo procurado então a busca continuará na metade posterior deste vetor. Caso o valor do meio vier depois do elemento procurado a busca continua na parte anterior a ele no vetor.

Exemplo de algoritmo de busca binária (em C):

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;

}

**Busca Sequencial**

Este algoritmo de busca é o mais simples algoritmo de busca que existe, já que consiste apenas em buscar o elemento no vetor percorrendo-o do inicio ao fim para achar o valor desejado. A complexidade desse algoritmo depende do caso, se for o melhor caso em que o valor buscado está no inicio do vetor a complexidade é O(1), se for o caso médio O((n+1)/2) e o pior caso é O(N) pois o valor estará na última posição do vetor.

Este algoritmo possui a vantagem por cima da busca binária por poder ser utilizado em vetores que não estão ordenados, pois a ordem não é relevante no algoritmo, mas a desvantagem é a sua demora caso o array em que for efetuado a busca esteja ordenado.

O algoritmo consiste basicamente em percorrer o array usando um for ou while e verificando com um if se o valor é igual ao que está sendo buscado ou não.

Exemplo de algoritmo de busca sequenciam (em C):

int buscaSequencial(int \*vetor, int chave, const int TAMANHO){

int i = 0;

while(i < TAMANHO && chave > vetor[i])

i++;

if(i < TAMANHO && chave == vetor[i])

return i;

else

return -1;

}