**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:

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.

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, 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)

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 numero 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.

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 };

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