---------------------------------------------------- P1 -----------------------------------------------

Pesquisa sequencial faz a busca de valor em valor, sequencialmente,

portanto não precisa ser ordenada.

Pesquisa binária precisa ser ordenada, uma vez que divide-se o vetor no meio

e refaz-se a busca, analisando se o valor é maior ou menor que o meio.

Descarta-se a metade não utilizada, e recomeça o processo

(divide>analisa>exclui), até achar o valor.

(posi+posf)/2

Para ordenar um vetor, cria-se um for encadeado, onde o primeiro vetor

vai de 0 a (N-1), sendo N o número do vetor. E o segundo vetor vai de 0 a N.

Se o primeiro vetor > segundo vetor, auxiliar = segundo vetor.

Segundo vetor = primeiro vetor. Primeiro vetor = auxiliar.

Selection sort troca o valor menor pelo primeiro do vetor. Ou seja, compara-se

o primeiro valor com os demais, trocando sempre que for maior. Ao chegar ao

fim do vetor, coloca-se o menor na primeira posição. Em seguida, recomeça-se

o ciclo a partir da segunda posição. Ao final, estarão todos ordenados.

Bubble sort faz a comparação de dois em dois elementos (posições do vetor),

sempre trocando de lugar deixando o menor na posição da esquerda.

Nota Final: 8,5

---------------------------------------- P1 ----------------------------------------

* **MERGE SORT**

Dividir para conquistar.

Primeiramente divide-se: Somando os índices do vetor, e divide por dois, arredondando para menos, sempre que necessário, dividindo-o em seguida.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | | | | | | 1 | | | | 2 | | | | 3 | | | | 4 | | | | | | 5 | | | | |
| 38 | | | | | | 16 | | | | 27 | | | | 39 | | | | 12 | | | | | | 27 | | | | |
| 0 | | | 1 | | | | | 2 | | | |  | | | 3 | | | | 4 | | | | 5 | | | |
| 38 | | | 16 | | | | | 27 | | | |  | | | 39 | | | | 12 | | | | 27 | | | |
| 0 | | | | 1 | | |  | | 2 | |  | | | | | 3 | | | | | 4 | | | |  | 5 | |
| 38 | | | | 16 | | |  | | 27 | |  | | | | | 39 | | | | | 12 | | | |  | 27 | |
| 0 |  | 1 | | |  | | | | | | | | 3 | | | |  | | | 4 | |
| 38 |  | 16 | | |  | | | | | | | | 39 | | | |  | | | 12 | |

O segundo passo é conquistar: Compara-se o segundo vetor da esquerda, com o primeiro. Se for menor, troca de lugar, e agrupa os subvetores.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0 | 1 |  | 3 | | 4 |
| 16 | 38 |  | 12 | | 39 |
|  |  |  |  | |  |
| 0 | 1 | 2 | |  | 3 | 4 | 5 |
| 16 | 27 | 38 | |  | 12 | 27 | 39 |

Na hora de agrupar os subvetores, compara –se o primeiro elemento da esquerda, com o primeiro elemento da direita. O menor, vai para a primeira posição do vetor original. Repete-se a comparação, até que a incrementação seja feita para o próximo espaço do vetor.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 |
| 12 | 16 | 27 | 27 | 38 | 39 |

CÓDIGO:

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

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

**if**(posicaoInicio == posicaoFim) **return**;

*// ordenacao recursiva das duas metades*

metadeTamanho = (posicaoInicio + posicaoFim ) / 2;

mergeSort(vetor, posicaoInicio, metadeTamanho);

mergeSort(vetor, metadeTamanho + 1, posicaoFim);

*// intercalacao no vetor temporario t*

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 ) { *// i passou do final da primeira metade, pegar v[j]*

vetorTemp[k] = vetor[j];

j++;

k++;

}

**else** {

**if** (j == posicaoFim + 1) { *// j passou do final da segunda metade, pegar v[i]*

vetorTemp[k] = vetor[i];

i++;

k++;

}

**else** {

**if** (vetor[i] < vetor[j]) {

vetorTemp[k] = vetor[i];

i++;

k++;

}

**else** {

vetorTemp[k] = vetor[j];

j++;

k++;

}

}

}

}

*// copia vetor intercalado para o vetor original*

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

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

}

free(vetorTemp);

* **QUICK SORT**

Ordenação por troca de partição. Dividir para conquistar.

Primeiro escolhe o pivô (elemento qualquer no vetor, porém quanto mais mediano for o valor do pivô, melhor será a divisão dos dados). Em seguida particiona os dados, de forma a deixar os menores elementos do lado esquerdo do pivô, e os maiores ao lado direito.

Escolhe-se o pivô como o pivô como o mediano.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 4 | 2 | 5 | 1 | 6 | 3 | 7 |

Uma variável para primeiro elemento (p), e uma para último elemento do vetor (u).

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| P |  |  |  |  |  | u |
| 4 | 2 | 5 | 1 | 6 | 3 | 7 |

Sempre que a primeira variável for menor ou igual ao pivô, é incrementada.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | p |  |  |  |  | u |
| 4 | 2 | 5 | 1 | 6 | 3 | 7 |

Caso ela seja maior que o pivô, é congelada e começamos a usar a última variável.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  | P |  |  |  | u |
| 4 | 2 | 5 | 1 | 6 | 3 | 7 |

Se o valor da variável for maior que o pivô, essa é decrementada.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  | p |  |  | U |  |
| 4 | 2 | 5 | 1 | 6 | 3 | 7 |

Quando o valor da última variável for menor que o pivô, trocamos ela de lugar com a primeira variável.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  | p |  |  | U |  |
| 4 | 2 | 3 | 1 | 6 | 5 | 7 |

E voltamos a usar a primeira variável.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  | P |  | U |  |
| 4 | 2 | 3 | 1 | 6 | 5 | 7 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  | P | U |  |
| 4 | 2 | 3 | 1 | 6 | 5 | 7 |

Até que a primeira variável seja maior que o pivô novamente. Congelamos ela novamente, e voltamos a última variável.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  | P U |  |  |
| 4 | 2 | 3 | 1 | 6 | 5 | 7 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  | U | P |  |  |
| 4 | 2 | 3 | 1 | 6 | 5 | 7 |

Quando a última variável ultrapassar a primeira, trocamos o pivô de lugar com ela. O pivô estará ordenado.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 6 | 5 | 7 |

Recomeça o particionamento com os vetores que foram gerados (antes e depois do pivô).

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| P |  | U |  |  |  |  |
| 1 | 2 | 3 | 4 | 6 | 5 | 7 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | P | U |  |  |  |  |
| 1 | 2 | 3 | 4 | 6 | 5 | 7 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | P U |  |  |  |  |  |
| 1 | 2 | 3 | 4 | 6 | 5 | 7 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| U | P |  |  |  |  |  |
| 1 | 2 | 3 | 4 | 6 | 5 | 7 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 6 | 5 | 7 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | P | U |  |  |  |  |
| 1 | 2 | 3 | 4 | 6 | 5 | 7 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  | P U |  |  |  |  |
| 1 | 2 | 3 | 4 | 6 | 5 | 7 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | U | P |  |  |  |  |
| 1 | 2 | 3 | 4 | 6 | 5 | 7 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 6 | 5 | 7 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 6 | 5 | 7 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  | P |  | U |
| 1 | 2 | 3 | 4 | 6 | 5 | 7 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | P | U |
| 1 | 2 | 3 | 4 | 6 | 5 | 7 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  | P U |
| 1 | 2 | 3 | 4 | 6 | 5 | 7 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | U | P |
| 1 | 2 | 3 | 4 | 6 | 5 | 7 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |

CÓDIGO:

#include*<stdio.h>*

#define TAM 10

void quick(int vet[], int esq, int dir){

int pivo = esq,i,ch,j;

**for**(i=esq+1;i<=dir;i++){

j = i;

**if**(vet[j] < vet[pivo]){

ch = vet[j];

**while**(j > pivo){

vet[j] = vet[j-1];

j--;

}

vet[j] = ch;

pivo++;

}

}

**if**(pivo-1 > esq){

quick(vet,esq,pivo-1);

}

**if**(pivo+1 < dir){

quick(vet,dir,pivo+1);

}

}

int main(){

int vet[TAM],i;

**for**(i=0;i<TAM;i++)

scanf("%d",&vet[i]);

quick(vet,0,TAM-1);

**for**(i=0;i<TAM;i++)

printf("%d ",vet[i]);

printf("**\n**");

**return** 0;

}

* **RADIX SORT**

CÓDIGO:

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;

}

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 170 | 45 | 75 | 90 | 2 | 24 | 802 | 66 |

Ordenando pelo último digito (unidade)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 170 | 90 | 2 | 802 | 24 | 45 | 75 | 66 |

Ordenando o próximo dígito (dezena)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 02 | 802 | 24 | 45 | 66 | 170 | 75 | 90 |

Ordenando pelo próximo dígito (centena)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 002 | 024 | 045 | 066 | 075 | 090 | 170 | 802 |

* **ARVORE AVL**

Árvore AVL (ou árvore balanceada pela altura), é uma [árvore](https://pt.wikipedia.org/wiki/%C3%81rvore_(estrutura_de_dados)) de busca binária. Em tal árvore, as alturas das duas sub-árvores a partir de cada nó diferem no máximo em uma unidade. As operações de busca, inserção e remoção de elementos possuem complexidade{\displaystyle O(\log n)}. Inserções e remoções podem também requerer o rebalanceamento da árvore, exigindo uma ou mais rotações.

FB(n) = altura(sadireita) – altura(saesquerda).

• se FB(n) = 0, as duas sub-árvores têm a mesma altura;

• se FB(n) = -1, a sub-árvore esquerda é mais alta que a direita em 1;

• se FB(n) = +1, a sub-árvore direita é mais alta que a esquerda em 1

♣ Resolvemos o desbalanceamento efetuando rotações

• Simples ¬Sinais (fb) são iguais

• Dupla ¬Sinais (fb) são diferentes

• Direita ¬FB (+) positivo

• Esquerda ¬FB (-) Negativo

* **ARVORE B-TREE**

Cada nó tem um número variável de chaves e filhos

♣ As chaves são armazenadas em ordem crescente

♣ Cada chave tem um filho associado que é o root da sub-árvore contendo todos os nós com chave menor ou igual a chave,mas maior que a chave predecessora

♣ Um nó também tem o filho mais a direita sendo o root da subárvore contendo todas as chaves maiores que qualquer chave no nó

**Definição**

As árvores B são árvores balanceadas projetadas para trabalhar com dispositivos de armazenamento secundário como discos magnéticos.

Elas visam otimizar as operações de entrada e saída nos dispositivos. O tempo de acesso às informações em um disco é prejudicado principalmente pelo tempo de posicionamento do braço de leitura. Uma vez que o braço esteja posicionado no local correto, a leitura pode ser feita de forma bastante rápida. Desta forma, devemos minimizar o número de acessos ao disco.

Diferente das árvores binárias, cada nó em uma árvore B pode ter muitos filhos, isto é, o grau de um nó pode ser muito grande.

Definição: Uma árvore B possui as seguintes propriedades:

1. Todo o nó X possui os seguintes campos:
   1. n, o número de chaves armazenadas em X;
   2. as n chaves k1, k2...kn são armazenadas em ordem crescente;
   3. folha, que indica se X é uma folha ou um nó interno.
2. Se X é um nó interno então ele possui n+1 ponteiros f1, f2...fn+1 para seus filhos (podendo alguns serem nulos)
3. Se ki é alguma chave na sub-árvore apontada por fi, então
4. Todas as folhas da árvore estão na mesma altura (que é a altura da árvore).
5. Existe um número máximo e mínimo de filhos em um nó. Este número pode ser descrito em termos de um inteiro fixo t maior ou igual a 2 chamado grau mínimo.
   1. Todo o nó diferente da raiz deve possuir pelo menos t-1 chaves. Todo o nó interno diferente da raiz deve possuir pelo menos t filhos. Se a árvore não é vazia, então a raiz possui pelo menos uma chave.
   2. Todo o nó pode conter no máximo 2t - 1 chaves. Logo um nó interno pode ter no máximo 2t filhos. Dizemos que um nó é cheio se ele contém 2t - 1 chaves.

**Estrutura do Nó**

const t = 2;

typedef struct no\_arvoreB arvoreB;

struct no\_arvoreB {

int num\_chaves;

char chaves[2\*t-1];

arvoreB \*filhos[2\*t];

bool folha;

};