Classificação por Seleção - selection sort

Com base no que foi visto, construa um módulo, que receba como parâmetros um vetor de inteiros e o número de elementos neste vetor. Este módulo deve ordenar o vetor implementando a classificação por seleção.

UNIVASF

```c
void selecao(int vet[], int n) {
    int aux, j, i, maior;
    for (i = 0; i < n - 1; i++) {
        maior = 0;
        for (j = 1; j < n - i; j++)
            if (vet[j] > vet[maior])
                maior = j;
        /* Troca vet[n-i-1] com vet[maior]*/
        aux = vet[n - i - 1];
        vet[n - i - 1] = vet[maior];
        vet[maior] = aux;
    }
}
```

UNIVASF

Classificação por Inserção - insertion sort

- Método preferido dos jogadores de **cartas**.
- O jogador vai recebendo cartas uma por uma, e inserindo-as na posição adequada em sua mão, fazendo com que as cartas permaneçam ordenadas durante todo o jogo.
- ou seja,
  - Em cada passo, a partir de i=2, o i-ésimo item da sequência fonte é apanhado e transferido para a sequência destino, sendo inserido no seu lugar apropriado.

UNIVASF

Classificação por Inserção - insertion sort

A proposta da ordenação por inserção é a seguinte:

Para cada elemento do vetor **faça**
    Inseri-lo na posição que lhe corresponde;

Um processo *in situ,* chamado **inserção direta**, pode ser assim descrito:

Considerar o subvetor ordenado v[1..k]
Para j de k+1 até n faça
    Inserir v[j] na sua posição em v [1..k]

UNIVASF

Classificação por Inserção - insertion sort

Para encontrar o lugar de v[k], basta comparar as chaves de índices [1..k-1] até encontrar uma chave v[k_ins] que seja maior que ele. v[k_ins] será a sucessora de v[k] depois deste ser inserido no subvetor ordenado (i.e., depois de ser localizado).

O problema que surge agora é: como arranjar lugar em v[1..k - 1] para o valor v[k]?

Bem, como v[k] vai deixar seu lugar, este pode ser ocupado pelo elemento v[k-1], ao se fazer avançar uma posição para frente todo o subvetor v[k_ins..k – 1].

UNIVASF

Classificação por Inserção - insertion sort

O processo começa adotando-se o subvetor ordenado v[1..1], e passando-se a inserir os elementos de índice 2..n.

Para melhorar um pouco o desempenho, inicia-se a pesquisa a partir da posição k, ao mesmo tempo que se vai deslocando os elementos v[k_ins..k - 1].

A busca da posição de inserção deve prosseguir enquanto v[i $\in$ 1..k-1] > v[k].

Se a chave v[k] é menor que qualquer chave v[1.. k -1], acaba-se percorrendo o vetor até o seu início, correndo-se o risco de um índice inválido (0). Por isso, o critério de parada deve incluir uma segunda condição: k_ins >= 1.

UNIVASF

Classificação por Inserção - insertion sort

Observe no exemplo a seguir o comportamento de um vetor submetido á classificação por inserção direta. Em cada linha é listado um vetor depois de uma passagem completa sobre o mesmo, estando sublinhado o subvetor [1..k-1] e em negrito o elemento nele inserido por último.

Vetor original: 75 25 95 87 64 59 86 40 16 49

passagem	**vetor resultante**
1	**25** 75 95 87 64 59 86 40 16 49

Classificação por Inserção - insertion sort

passagem	vetor resultante
2	25 75 **95** 87 64 59 86 40 16 49
3	25 75 **87** 95 64 59 86 40 16 49
4	25 **64** 75 87 95 59 86 40 16 49
5	25 **59** 64 75 87 95 86 40 16 49
6	25 59 64 75 **86** 87 95 40 16 49
7	25 **40** 59 64 75 86 87 95 16 49
8	**16** 25 40 59 64 75 86 87 95 49
9	16 25 40 **49** 59 64 75 86 87 95

Com base no que foi discutido, codifique um módulo que receba como parâmetros um vetor (de inteiros) e o número de elementos no mesmo. Através da inserção direta o módulo deve ordenar de forma crescente os elementos contidos no vetor.

UNIVASF

```c
void insercao_direta(int vet[], int n) {
    int k_ins, k, elemento;
    for (k = 1; k < n; k++) {
        k_ins = k - 1;
        elemento = vet[k];
        while (k_ins >= 0 && vet[k_ins] > elemento) {
            vet[k_ins + 1] = vet[k_ins];
            k_ins = k_ins - 1;
        }
        vet[k_ins + 1] = elemento;
    }
}
```

UNIVASF

Classificação por Particionamento - quicksort

O método de particionamento é um caso de aplicação do princípio da divisão e conquista: classificar dois vetores de tamanho n/2 é mais fácil que classificar um vetor de tamanho n.

O método basicamente se resume em:

i. particionar o vetor;
ii. classificar cada partição.

UNIVASF

Classificação por Particionamento - quicksort

Um processo que trabalha com base nessa proposta é o *quicksort*, assim denominado por seu inventor, C. A. R. Hoare.

É um processo *in situ* e seu caso trivial é a partição de tamanho 1.

Pode ser visto como uma melhoria do método das trocas, buscando reposicionar primeiro as chaves mais distantes do seu lugar final.

UNIVASF

Classificação por Particionamento - quicksort

Funda-se em separar o vetor em duas partições, delimitadas por uma certa chave, dita *pivô*, de forma que numa das partições estejam todas as chaves menores ou iguais ao pivô e na outra todas as chaves maiores que ele. Em seguida, reclassifica-se cada partição.

Para uma melhor compreensão vamos analisar um exemplo (consideraremos o primeiro elemento da partição como o pivô, com o objetivo de facilitar o processo).

UNIVASF

Classificação por Particionamento - quicksort

Se um vetor inicial for dado como:

25 57 48 37 12 92 86 33

e o pivô (25) for colocado na posição correta o vetor resultante será:

12 25 57 48 37 92 86 33

agora, o problema inicial foi decomposto na classificação de dois subvetores:

(12) e (57 48 37 92 86 33)

O primeiro já está classificado, uma vez que tem apenas um elemento. O processo se repete para classificar o segundo subvetor.

Classificação por Particionamento - quicksort

Agora, o vetor pode ser visualizado como:
12 25 (57 48 37 92 86 33)
onde os parênteses encerram os subvetores que ainda serão classificados. Repetindo o processo sobre o subvetor a ordenar teremos:
12 25 (48 37 33) 57 (92 86)
e as aplicações posteriores resultam em:
12 25 (37 33) 48 57 (92 86)
12 25 (33) 37 48 57 (92 86)
12 25 33 37 48 57 (92 86)
12 25 33 37 48 57 (86) 92
12 25 33 37 48 57 86 92

UNIVASF

Classificação por Particionamento - quicksort

Os observadores mais atentos devem ter percebido que o quicksort pode ser melhor compreendido se definido recursivamente.

É interessante observar que no método apresentado os elementos após o particionamento aparecem na mesma ordem relativa em que apareciam no vetor original.

Entretanto, este método de particionamento é ineficiente de se implementar. Veremos agora um método eficiente de se implementar o particionamento.

UNIVASF

Classificação por Particionamento - quicksort

Para tanto, com um cursor, digamos “esq”, percorre-se o vetor da esquerda para a direita, até se localizar um elemento maior que o pivô.

Com outro cursor, “dir”, percorre-se o vetor da direita para a esquerda, até se encontrar um elemento menor ou igual ao pivô, esses valores são intercambiados caso “dir” seja maior que “esq”.

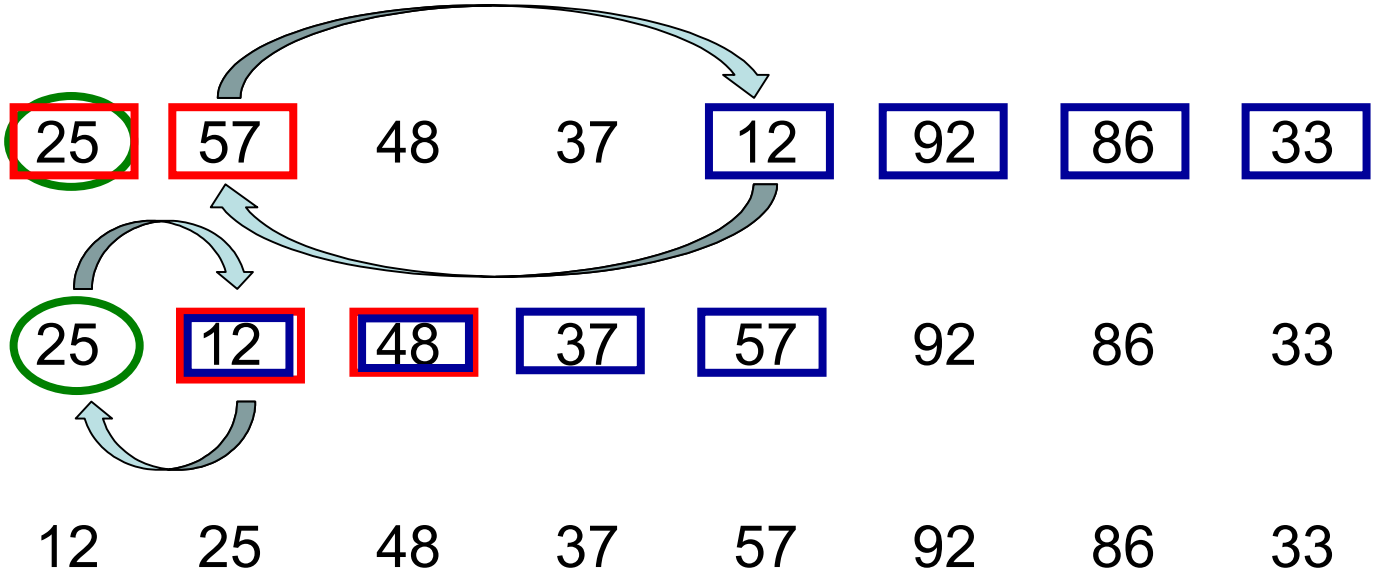
Repete-se este processo até os setores perscrutados esgotarem o vetor, o que se pode detectar pelo encontro dos cursores, ou seja, quando esq > dir.

Neste momento, troca-se o valor do pivô pelo valor do elemento indexado por dir que é dito “ponto de partição”. O qual delimitará as partições a serem reclassificadas.

UNIVASF

Classificação por Particionamento - quicksort

Este processo pode ser observado no exemplo:



Ponto de particionamento

Legenda: **Pivô** **V[esq]** **V[dir]**

Classificação por Particionamento - quicksort

Exercício:

Com base no que foi discutido implemente um módulo denominado particionar. O qual recebe, como parâmetros, um vetor de inteiros e os índices do limite inferior e superior de um subvetor pertencente a este e retorna o índice do ponto de particionamento.

UNIVASF

```
int particionar(int v[], int ii, int is) {
    int esq = ii, dir = is, pivo = v[ii], aux;
    while (esq < dir) {
        while (v[esq] <= pivo && esq < is) {
            esq++;
        }
        while (v[dir] > pivo) {
            dir--;
        }
        if (esq < dir) {
            aux = v[esq];
            v[esq] = v[dir];
            v[dir] = aux;
        }
    }
    v[ii] = v[dir];
    v[dir] = pivo;
    return dir;
}
```

UNIVASF

Classificação por Particionamento - quicksort

Exercício:

Agora, construa um módulo recursivo que receba, como parâmetros, um vetor de inteiros e os índices do primeiro e do último elemento contidos no vetor. O módulo em questão deve ordenar o vetor implementando o método quicksort.

UNIVASF

```
int particionar(int v[], int ii, int is) {
    int esq = ii, dir = is, pivo = v[ii], aux;
    while (esq < dir) {
        while (v[esq] <= pivo && esq < is) {
            esq++;
        }
        while (v[dir] > pivo) {
            dir--;
        }
        if (esq < dir) {
            aux = v[esq];
            v[esq] = v[dir];
            v[dir] = aux;
        }
    }
    v[ii] = v[dir];
    v[dir] = pivo;
    return dir;
}
```

UNIVASF

```
void quicksort(int v[], int ind_pri_ele, int ind_ult_ele) {
    int pont_part;
    if (ind_ult_ele - ind_pri_ele > 0) {
        pont_part = particionar(v, ind_pri_ele, ind_ult_ele);
        quicksort(v, ind_pri_ele, pont_part - 1);
        quicksort(v, pont_part + 1, ind_ult_ele);
    }
}
```

UNIVASF

Classificação por Particionamento - quicksort

A tarefa de escolher o pivô pode ser executada de forma mais eficiente.

A chave ideal para o pivô seria a mediana das chaves, com a qual se teria uma divisão a mais balanceada possível.

Só que para determinar a mediana é preciso ordenar o vetor!

Como a distribuição das chaves é, em princípio, aleatória, qualquer uma tem a mesma chance de ser mediana.

Mas, ao se tomar a primeira, como foi feito anteriormente, corre-se o risco de ser esta a menor (ou a maior) de todas, tornando praticamente inócuo o trabalho na primeira fase de particionamento (pois se teria uma partição sem nenhum elemento e outra com *n-1* elementos).

Uma escolha melhor é a mediana entre a primeira chave, a última e a do meio do vetor.

UNIVASF

Classificação por Particionamento - quicksort

Exercício:

Reescreva os algoritmos anteriores, considerando que o pivô não é mais o primeiro elemento mas, sim a mediana entre o primeiro elemento, o último e o elemento do meio do vetor.

UNIVASF

Métodos de Pesquisa

Objetivos e Caracterizações

Para que se possa falar em algoritmos de pesquisa, é necessário inicialmente introduzir a noção de mapeamento que é uma das mais primitivas em programação. Refere-se a uma regra de associação entre os valores de um conjunto (domínio) e os valores de outro (imagem).

Escreve-se

$$\mathbf{m}: \mathbf{S} \rightarrow \mathbf{T}$$

para se declarar que **m** é um mapeamento do conjunto **S** para o conjunto **T**. Obtém-se um valor de **T** aplicando-se **m**(**i**), onde $\mathbf{i} \in \mathbf{S}$.

Os vetores e matrizes são os casos típicos. Onde os índices são normalmente objetos simples ou no máximo tuplas homogenias (pares, triplas, etc.) de valores simples.

UNIVASF

Objetivos e Caracterizações

As estruturas chamadas **tabelas** são a realização da idéia genérica de mapeamento, em que os valores do domínio podem ser quaisquer. A organização das tabelas pode se reduzir aos arranjos, mas para se falar em tabelas, usa-se uma terminologia específica:

tabela: uma coleção de *entradas*;

entrada: um conjunto de *campos*, formando um *registro*, ou *linha*, da tabela;

chave: um campo escolhido para *identificar* a entrada.

Como pode-se perceber uma operação importante é a busca de uma entrada dado o valor da chave.

UNIVASF

Objetivos e Caracterizações

A tabela, como mapeamento, poderia ser operada, por exemplo, para uma consulta, da seguinte forma:

E = T[C];

Considerando E: entrada, T:tabela; C:chave.

Diversas estratégias são propostas para implementação da operação de pesquisa, levando em conta aspectos das operações usuárias e da representação física da tabela. Veremos agora dois métodos utilizados para implementação de pesquisa em tabelas, a pesquisa sequencial e a pesquisa binária.

UNIVASF

Pesquisa Sequencial

A pesquisa sequencial é o método mais simples.

Consiste na mera varredura serial, entrada por entrada, devolvendo-se o índice da entrada cuja chave for igual à chave fornecida como argumento da pesquisa. Ou devolvendo 0 (zero), convencionalmente, caso a chave buscada não seja localizada, tendo-se comparado todas as chaves, até o fim da tabela.

O algoritmo (módulo) a seguir nos mostra uma função que efetua uma busca sequencial em uma tabela.

UNIVASF

```
int pesq(Tabela T, Chave C) {
    int i;
    for (i = 0; i < T.N; i++) {
        if (T.TAB[i].CH == C) {
            return (i+1);
        }
    }
    return 0;
}
```

UNIVASF

Pesquisa Sequencial

O desempenho da pesquisa sequencial pode melhorar um pouco se a tabela estiver *ordenada* em função da chave: pode-se interromper a pesquisa assim que se alcançar uma entrada com chave maior do que a pesquisada (no caso de uma ordenação crescente), significando ser desnecessário prosseguir até o fim da tabela.

O pior caso (busca da última chave) continua gerando uma varredura total da tabela.

Exercício: Construa um módulo que efetue uma busca sequencial em uma tabela ordenada de forma crescente em função da chave.

UNIVASF

```c
int pesq_seq(Tabela T, Chave C) {
    int i;
    for (i = 0; i < T.N; i++) {
        if (T.TAB[i].CH >= C) {
            if (T.TAB[i].CH == C) {
                return (i+1);
            } else {
                return 0;
            }
        }
    }
    return 0;
}
```

UNIVASF

Pesquisa Binária

Sobre a tabela ordenada é possível se obter um algoritmo bem mais eficiente.

A ideia do mesmo pode ser assimilada pelo seguinte exemplo: ao procurar uma palavra no dicionário (que é uma tabela ordenada), começa-se em qualquer ponto do mesmo.

Se a página aberta contiver a palavra, a busca terminou;

Senão, a palavra pode estar antes ou depois dessa página, conforme ela seja lexicograficamente menor ou maior que as palavras dessa página.

UNIVASF

Pesquisa Binária

De modo que novamente se abre o dicionário no setor adequado, anterior ou posterior, repetindo-se este processo até encontrar a palavra.

Exercício: No algoritmo que elaboraremos agora, a primeira comparação de chaves é feita no meio da tabela. Se não for encontrada aí a chave procurada, pode-se abandonar metade da tabela, repetindo o processo com a divisão da outra metade pelo meio, até ser encontrada a chave ou ter-se uma metade constituída de apenas uma entrada, caracterizando-se assim a ausência da chave na tabela.

UNIVASF