Possível resolução / principais dificuldades

Alínea A:

- Ler uma string - Para ler a string, e atendendo a que cada string é colocada numa linha, o mais simples é utilizar o gets, e quem não quiser o warning utiliza como já terá feito para as atividades formativas, o fgets (dado no módulo 3). Esta função tem por objetivo precisamente este, ler uma string até ao final da linha. O scanf pode ser utilizado, e foi por muitos estudantes, mas tem por objetivo ler dados formatados, por exemplo, dois números inteiros.
- Caracteres entre A e Z - Teriam de fazer um ciclo para testar as 10 letras. Este teste pode ser feito apenas se a string tiver 10 caracteres, mas têm de fazer 10 testes, pelo que é apropriado um ciclo.

Possível resolução

```
#define BUFFER 100
void main()
{
    char str[BUFFER];
    int i;
    fgets(str,BUFFER,stdin);
    str[strlen(str)-1]=0; // remover o último carater, fim-de-linha
    if(strlen(str)!=10) {
        printf(""0"");
        return;
    }

    for(i=0;i<10;i++)
        if(str[i]<'A' || str[i]>'Z') {
            printf(""0"");
            return;
        }
    printf(""1"");
}
```

Possível explicação no relatório para este código:

- Foi utilizado o vetor "str" para ler a string de entrada, não sendo utilizadas funções.
- Lida a string, é verificado se tem 10 caracteres, retornando 0 caso contrário, e se todos os caracteres estão entre 'A' e 'Z', retornando 1 nesse caso, ou 0 assim que um caracter esteja fora do intervalo.

Principais dificuldades:

- identificar o caracter a testar, e número de testes. Têm de conseguir aceder a todos os caracteres da string, neste caso seria str[i]

- colocação de texto não solicitado, para pedir ao utilizador a introdução da string, quando não era solicitado isso, apenas retornar 1 ou 0, tal como indicado nos casos de teste
- percorrer os caracteres da string, mas não parar assim que obter uma situação de falha, retornando apenas o resultado do último caracter
- Em vez de utilizar uma expressão, colocar as letras do alfabeto num vetor, para testar se cada caracter é igual a um dos elementos no vetor. Esta situação gera código ineficiente, e não faz uso do facto dos caracteres terem os códigos seguidos.
- Calcular quantas letras maiúsculas a string tem, comparando com 10. Este teste é distinto do solicitado, embora passe todos os casos de teste existentes. Evidentemente que é simples de fazer um caso de teste que falhe, basta fazerem uma string de 11 caracteres, com uma letra minúscula e as restantes todas maiúsculas, passa neste caso de teste, mas não passa no teste solicitado, dado que não tem tamanho 10.
- Existiu também a versão do condicional testes em 10 caracteres na mesma expressão, ficando naturalmente uma expressão bem longa (e código pouco generalizável, aplicável o EQ9).

Alínea B

- Importante colocar a alínea A numa função, dado que é uma funcionalidade que irá ser chamada para cada string introduzida.
- Ponto principal: como não podiam ir construindo o output antes de ler todos os dados de entrada, tinham de ter um local para guardar as strings válidas.
- estruturas utilizadas: vetor de strings, ou seja, vetor bidimensional de caracteres
- estrutura alternativa: uma só strings, a cada 10 caracteres inicia uma nova string (possível porque todas as strings têm 10 caracteres)

Possível resolução

```
#define BUFFER 100
#define MAXVECTOR 50
int StringValida(char *str) {
    int i;
    if(strlen(str)!=10)
        return 0;
    for(i=0;i<10;i++)
        if(str[i]<'A' || str[i]>'Z')
            return 0;
    return 1;
}

void main()
{
    char strValidas[MAXVECTOR][BUFFER];
    char str[BUFFER];
```

```
    int i, validas=0;
    while(fgets(str,BUFFER,stdin)!=NULL) {
        str[strlen(str)-1]=0;
        if(StringValida(str))
            strcpy(strValidas[validas++],str);
    }
    printf(""%d\n"",validas);
    for(i=0;i<validas;i++)
        printf(""%s\n"",strValidas[i]);
}
```

Possível explicação no relatório:

- Para além da alínea A, foi utilizado o vetor de strings, "strValidas" com todas as strings válidas lidas, e um contador "validas".
- Foi criada uma função StringValidas, que recebe uma string e retorna 0/1 de acordo com a alínea A.
- Primeiramente lê-se as strings uma a uma, até que não existam mais dados, e se a string lida é válida, esta é copiada para o vetor "strValidas", sendo incrementado o contador "validas".
- No final é apresentado o número de strings válidas lido, seguido das respetivas strings.

Principais dificuldades:

- Saber quando terminar - Têm de utilizar o valor de retorno das funções para saber se houve de facto algo lido, ou não há mais nada para ler. O gets/fgets retornam NULL se não lerem nada, o scanf retorna o número de argumentos lidos. Houve quem tivesse fixado um valor para o número de linhas (pior solução), ou uma marca (melhor solução), mas apenas se pararem quando não há mais dados de entrada, é que o programa funcionaria sem problema. A utilização de um vetor de strings com todas as strings lidas, de modo a dar ao gets/scanf sempre um endereço de memória não inicializado, de modo a detectar se a string foi lida ou não, funciona apenas se a memória estiver zerada. Não é de todo aconselhado. Outra forma foi ver se a string se mantinha igual, mas não concluindo/suspeitando que nesse caso a string não era lida. Não fazia de todo sentido, no caso de não conseguir ler uma string, colocar no buffer a última string lida, mas por observação alguns estudantes optaram por fazer este teste.
- Como o utilizador pode terminar? - Um utilizador poderia terminar a introdução de texto, indicando o fim da entrada de dados (no Linux Ctrl+D no Windows Ctrl+Z). Como o VPL está em Linux, devem fazer Ctrl+D. Mas esta questão é indiferente, seria útil apenas para produzirem casos de teste. Poderiam assumir no relatório um outro teste para terminar (introdução da string ".", por exemplo), desde que terminassem sempre assim que não consigam ler mais strings.
- Recursos VPL excedidos - Esta situação terá sido disparada apenas por excesso de tempo, sendo o principal motivo não saber quando parar a

leitura de strings, continuando o programa a ler quando já não há mais nada para ler.

- ter pelo menos 10 caracteres - Esta interpretação do enunciado, não compreendo qual a real origem. O enunciado é compreendido por vários estudantes desta forma, quando é referido na introdução do enunciado "todas de tamanho K" mais tarde "considere K=10", e "se a string tiver 10 caracteres". Não há nenhuma expressão "pelo menos 10 caracteres".
- verificar se a string tem 10 letras maiúsculas - Vários estudantes a fazerem este teste. Não é isso que é solicitado, é uma string com 10 caracteres, com todas as letras maiúsculas. Este teste pode até ser correto nos caso de teste utilizados, mas facilmente falhariam para strings com mais de 10 letras, mas com apenas 10 letras maiúsculas, devendo retornar 0 nessas strings, enquanto que este teste retorna 1.
- guardar todas as strings lidas - Não é necessário ter espaço para todas as strings lidas, e para todas as strings válidas, basta as strings válidas. Se não fazem nada com uma dada informação, então não vale a pena guardar. Muitos estudantes duplicam o vetor para guardar todas as strings, as lidas e as válidas.
- Dificuldade no gets - Houve esta referência como justificação para utilização do scanf. Que dificuldade? Se há algum tipo de dificuldade, devem colocar no fórum, e não guardar receios que depois podem vos levar a procurar evitar problemas que não existem, como o caso de não utilizar o gets para evitar problemas.
- Quem considera o VPL estranho e diferente de "outros IDEs" por ter lá as linhas já introduzidas Ver em Atividades 3 o capítulo de ficheiros, como redirecionar o stdin para ser recebido de um ficheiro, e se se quiser enviar o stdout para um ficheiro. O VPL apenas faz uso deste redireccionamento do stdin, e coloca também o stdout para um ficheiro, de modo a poder comparar com as respostas certas.

Alínea C

- Importante a reutilização de um dos exercícios de ordenação, alterado para ordenar um índice em vez do próprio vetor.
- Para normalizar as letras, é importante uma função para conter a complexidade e potenciar a sua reutilização na aliena seguinte. é preciso para cada posição, calcular a frequência, e mapear as letras de acordo com essa frequência.
- Na função de normalização, é necessário um vetor para contar a frequência das letras, e dois vetores um com as letras ordenadas pela frequência, e outro com o índice invertido para facilitar a substituição.
- A constatação da necessidade do índice invertido poderia ser observada ao tentar utilizar o índice com as letras ordenadas. Ao procurar saber que letra colocar se tiver uma posição com uma letra D, por exemplo, teria de saber a posição da letra D na lista ordenada de letras. Podia-se naturalmente percorrer o vetor de letras ordenadas à procura da letra D,

cada vez que esta é encontrada. Com o índice invertido este passo não é feito de uma só vez. Naturalmente que código ineficiente não é penalizado.

- Houve quem tivesse colocado num número, tanto a frequência da letra (com mais peso, multiplicado por 100, por exemplo), e o código da própria letra nos dígitos menos significativos (para garantir o segundo critério de ordenação). Este método tem também a vantagem, de se saber a que letra pertence cada elemento, é o resto da divisão por 100.
- A versão do código com a letra mais frequente, não precisa do índice invertido.

Possível resolução

```
#define BUFFER 100
#define MAXVECTOR 50
#define TAMSTRING 10
// código das AFs (modificado)
void Sort(int v[], int id[], int n)
{
   int i,j,aux;
    for(i=0;i<n;i++)
        for(j=i+1;j<n;j++)
            if(v[id[i]]<v[id[j]] ||
                v[id[i]]==v[id[j]] &&
                id[i]>id[j])
            {
                aux=id[i];
                id[i]=id[j];
                id[j]=aux;
            }
}
int StringValida(char *str) {
...
}
void NormalizarLetras(char strValidas[MAXVECTOR][BUFFER], int validas)
{
    int frequencia['Z'-'A'+1]; // frequencias
    int id['Z'-'A'+1]; // ordem das letras
    int invID['Z'-'A'+1]; // inverter o índice para mais facilmente aplicá-lo
    int i, j;

    for(i=0;i<TAMSTRING;i++) {
        for(j=0;j<'Z'-'A'+1;j++) {
            frequencia[j]=0;
            id[j]=j;
        }
        for(j=0;j<validas;j++)
            frequencia[strValidas[j][i]-'A']++;
        Sort(frequencia,id,'Z'-'A'+1);

        // inverter o índice
        for(j=0;j<'Z'-'A'+1;j++)
            invID[id[j]]=j; // o primeiro é o 0, seja qual for a letra, fica com a
```

```
sua posição
        // efetuar a troca utilizando invID
        for(j=0;j<validas;j++)
            strValidas[j][i]='A'+invID[strValidas[j][i]-'A'];

    }
}
void main()
{
    char strValidas[MAXVECTOR][BUFFER];
    char str[BUFFER];
    int i, validas=0;
    while(fgets(str,BUFFER,stdin)!=NULL) {
        str[strlen(str)-1]=0;
        if(StringValida(str))
            strcpy(strValidas[validas++],str);
    }
    NormalizarLetras(strValidas, validas);

    printf(""%d\n"",validas);
    for(i=0;i<validas;i++)
        printf(""%s\n"",strValidas[i]);
}
```

Possível explicação no relatório

- Explicar as estruturas de dados (em cima).
- A função NormalizarLetras recebe um conjunto de strings válidas, e normaliza todas as strings de acordo com o solicitado.
- São realizados os seguintes passos para cada índice de letra:
    - inicializar vetor "frequencia" e "id", seguido do calculo das frequências.
    - É chamada a ordenação das letras por frequência, sendo devolvido um índice, que é de seguida invertido, de modo a poder-se substituir um dado caracter c por o índice invertido, ou seja, c='A'+invID[c-'A'].
    - Notar que c irá ser a letra 'A' caso a posição de ordenação (índice invertido) de c for 0 (o primeiro), qualquer que seja a letra c. Será a letra 'Z' no caso da posição de ordenação de c for 'Z'-'A' (o último), qualquer que seja o valor de c.

As principais dificuldades nesta e na alínea seguinte, são muito diversificadas, e por vezes não é clara a dificuldade com base no e-fólio. Por um lado seria fastidioso enumerar as dificuldades, por outro lado, seria complicado explicar uma dificuldade para um público alargado. Aconselha-se aos estudantes que não têm esta e a alínea seguinte completa, a terminarem as alíneas, expondo as dificuldades no fórum se necessário.

Existe no entanto uma dificuldade standard, que irá ocorrer com muito maior frequência no e-fólio B, e com resultados mais nefastos. A não inicialização de variáveis, irá fazer com que o programa possa funcionar às vezes (assumindo que a memória não utilizada está zerada), e não em outras vezes (quando a memória não utilizada não está zerada). Como resultado o programa pode ter resultados distintos para os mesmos dados de input, ou crashar em local desconhecido. Evidentemente que o

crash, tal como num acidente rodoviário, é provocado num local, por exemplo, excesso de velocidade numa curva, existindo piso escorregadio, e continua a correr até que embate numa rocha, por exemplo. Naturalmente que não é a posição da rocha que é o problema, pelo que não devem simplesmente olhar para o local do crash, e abrir uma estrada a partir daí para que o carro não embata na rocha. Devem sim, identificar o local em que sai de pista, e remover o problema de raiz.

Relativamente ao problema anterior, é normal a atribuição de culpas no VPL (utiliza o GCC). A utilização de "culpas" tem como objetivo remover as responsabilidades do programador. É o compilador, utilizado internacionalmente há muitos anos, por milhões de pessoas, que tem o problema, e não o seu próprio código. Deve portanto o mundo mudar, para que o seu programa funcione. O facto de considerarem esta possibilidade como sendo real, retiram tempo para encontrarem o real problema, já que a situação não é da sua responsabilidade. Devem sempre procurar resolver os problemas, com o campo de ação disponível, e se existir de facto um bug no GCC, deveriam ter forma de comprovar o bug com base no menor código possível que contém o bug.

Alínea D

- Importante identificação da função DistanciaStrings, que devolve a distância entre duas strings
- Também importante a função MenorDistancia, que devolve a menor distância de uma string ao conjunto de strings válidas.
- Finalmente a função principal, é importante colocar também numa função, neste caso MenorString, que implementa o algoritmo solicitado após a normalização das strings. Esta função consiste em executar os 10 passos (terminando mais cedo se não existirem alterações), considerando todas as strings válidas que geram a menor distância.
- É importante ter uma estrutura com as frequências das strings válidas selecionadas (as que geram a menor distância), para poder obter as letras mais frequentes em cada posição. No entanto, para a menor string só são utilizadas as letras mais frequentes cuja frequência seja de pelo menos M/4 (divisão inteira).

Possível resolução

```
void Sort(int v[], int id[], int n)
{
...
}
int StringValida(char *str) {
...
}
void NormalizarLetras(char strValidas[MAXVECTOR][BUFFER], int validas)
{
...
}
int DistanciaStrings(char *str1, char *str2)
{
    int resultado = 0;
    while (*str1 != 0) {
```

```
        if ((*str1) != (*str2))
            resultado++;
        str1++;
        str2++;
    }
    return resultado;
}
int MenorDistancia(char strValidas[MAXVECTOR][BUFFER], int validas, char *str)
{
    int resultado = 0, i;
    for (i = 0; i<validas; i++)
        if (DistanciaStrings(str, strValidas[i])>resultado)
            resultado = DistanciaStrings(str, strValidas[i]);
    return resultado;
}
void MenorString(char strValidas[MAXVECTOR][BUFFER], int validas, char *menorString)
{
    int i, j, passos, distancia, M, alteracoes;
    int frequencia['Z' - 'A' + 1][TAMSTRING]; // frequencias nas diversas posições
    char maisFrequentes[BUFFER];
    strcpy(menorString, ""AAAAAAAAAA"");
    alteracoes = 1;
    for (passos = 0; passos<10 && alteracoes>0; passos++) {
        distancia = MenorDistancia(strValidas, validas, menorString);
        //printf(""\npasso %d, string %s. Distancia %d, M="",passos,menorString,
distancia);
        for (i = 0; i<'Z' - 'A' + 1; i++)
            for (j = 0; j<TAMSTRING; j++)
                frequencia[i][j] = 0;
        M = 0;
        for (i = 0; i<validas; i++)
            if (DistanciaStrings(menorString, strValidas[i]) == distancia) {
                M++;
                for (j = 0; j<TAMSTRING; j++)
                    frequencia[strValidas[i][j] - 'A'][j]++;
            }
        // obter as letras mais frequentes, inicializar com a primeira
        strcpy(maisFrequentes, ""AAAAAAAAAA"");
        for (i = 0; i<'Z' - 'A' + 1; i++)
            for (j = 0; j<TAMSTRING; j++)
                if (frequencia[maisFrequentes[j] - 'A'][j]<frequencia[i][j])
                    maisFrequentes[j] = i + 'A';
        // atualizar a string de teste
        alteracoes = 0;
        for (j = 0; j<TAMSTRING; j++)
            if (frequencia[maisFrequentes[j] - 'A'][j] >= M / 4) {
                if (maisFrequentes[j] != menorString[j]) {
                    menorString[j] = maisFrequentes[j];
                    alteracoes++;
                }
            }
        //printf(""%d, mais frequentes %s, %d alterações, string final
%s."",M,maisFrequentes, alteracoes, menorString);
    }
}
```

```
void main()
{
    char strValidas[MAXVECTOR][BUFFER];
    char str[BUFFER], menorString[BUFFER];
    int i, validas = 0;
    while (fgets(str, BUFFER, stdin) != NULL) {
        str[strlen(str) - 1] = 0;
        if (StringValida(str))
            strcpy(strValidas[validas++], str);
    }
    NormalizarLetras(strValidas, validas);
    MenorString(strValidas, validas, menorString);
    printf(""%d %s"", MenorDistancia(strValidas, validas, menorString), menorString);
}
```

Possível explicação no relatório: da mesma forma que nas alíneas anteriores, já indicado em cima o caso concreto. Fica aqui a ordem de importância pela qual devem explicar o relatório:

1 Funções utilizadas (abstrações funcionais criadas). Deve ser claro o que entra e sai de cada função, e o que a função faz.

2 Estruturas de dados, globais e locais em cada função - explicar apenas as principais estruturas de dados, naturalmente que variáveis iteradoras não faz sentido referir

3 Funcionamento global de cada função, focando no ponto principal. Deixar uma frase relativamente a inicializações de estruturas de dados, e concentrar na operação essencial em cada função.

4 Evitar leitura de código. Esta prática é como se fazer o ponto 3 em primeiro lugar, introduzindo 1 e 2 à medida que aparecem, e sem remover detalhes, lendo tudo. A utilidade de texto deste tipo é muito reduzida, já que normalmente é preferivel ler o código, em que se pode ver primeiro as funções/estruturas de dados existentes antes de ler cada função, e ler por alto zonas de pouca importância. Evidentemente que o trabalho é muito facilitado se não cometerem erros de qualidade de código.