Notas gerais

Na apresentação de casos de teste, devem ser feitos como se apresenta qualquer outro assunto, da forma mais clara e compacta possível, de modo a que se perceba com pouco tempo. Neste caso as tabelas com uma linha por caso de teste é apropriado mediante o que pretendem apresentar. PrintScreens é não apresentar nada, apenas registar uma prova de que fizeram, não iriam certamente apresentar PrintScreens a um cliente, nem tão pouco vídeos com os testes.

Alínea A

Para separar o número do nome, dado que era identificado o separador "-" poderiam utilizar o strchr para localizar as ocorrências do separador, ou o strtok, ou fazer uma função que leia o número até ao próximo separador "-".

Incorreto é assumir que o número da UC têm sempre o mesmo número de caracteres, e simplesmente subtrair o número de caracteres correspondente aos casos de teste, neste caso 9, já que isso é uma opção específica (EQ19).

Alínea B

Daria para ler a string com o scanf normalmente, utilizando "%d - %d - %d - %d - %d - %s" já que o nome das AFs nos casos de teste não tinham espaços.

Houve quem construísse uma string com todos os números de UCs, verificando se uma UC existia no caso de ser uma substring (utilizando strstr). Este método tem dois problemas principais. Primeiro trata números em strings, desperdiçando recursos computacionais. Por outro, pode existir números de UCs com número de dígitos distintos, e poderá levar a falsos positivos, no caso de um número de UC estar contido em outro número de UC.

Alínea C / D

O nome das UCs deveria ser também alocado dinamicamente.

Alguns estudantes fizeram código para cada alínea, tendo-se de comentar/descomentar código para que os casos de teste funcionem bem para cada alínea. Deveriam ter utilizado condicionais. Assim obtêm apenas 50 pontos no HR por cada uma dessas alíneas, em vez de 100.

Alguns estudantes alocaram memória para armazenar um número fixo de UCs e AFs ou após conhecer o número total desses elementos. Esta estratégia não é verdadeiramente alocação dinâmica de memória. A alocação dinâmica de memória não se limita à alocação de memória para um vetor quando é conhecido o seu tamanho, deve permitir alocar memória quando é necessário criar um novo elemento e libertar memória quando é necessário destruir um elemento.

Em termos de resolução aconselhada, esperava-se o seguinte:

Estrutura de dados:

```
typedef struct SActividade
{
    char *nome; // nome da atividade
    int inicio, fim; // período de estudo em que está disponível e
final
    int realizada; // 1 se a atividade é realizada, 0 se ainda não, e
-1 se foi processada e não pode ser realizada
    int sessoes; // número de sessões de estudo necessárias
    struct SActividade *seg;
} TActividade;

typedef struct SUC
{
    int  numero;
    char *nome; // nome da UC
    float realizada; // percentagem da UC realizada
    TActividade *atividades;
    struct SLista *seg;
} TUC;
```

O próprio enunciado revela as estruturas de dados necessárias. Não há motivo para qualquer outra estrutura satélite ou dependente. Sobre estas estruturas é de prever a inclusão de métodos de manipulação, que permitem o resto do código abstrair-se de detalhes de memória, e típicas em qualquer tipo de dados abstrato, mas naturalmente poderiam ser outras as primitivas:

```
/* Adicionar uma nova atividade à lista */
void ActAdiciona(TUC *lista, char *nome, int inicio, int fim, int
realizada, int sessoes);
/* inverte a lista */
TActividade  *ActInverte(TActividade  *lista);
/* remover o primeiro elemento da lista, retornando para o segundo */
TActividade *ActRemove(TActividade *lista);
/* Adicionar uma nova UC à lista */
TUC *UCAdiciona(TUC *lista, int numero, char *nome);
/* remover o primeiro elemento da lista, retornando para o segundo */
TUC *UCRemove(TUC *lista);
/* retorna o elemento com o número de UC especificado, ou NULL */
TUC *UCElemento(TUC *lista, int numeroUC);
/* retorna o k-éssimo elemento, ou NULL */
TUC *UCkElemento(TUC *lista, int k);
/* insere de forma  ordenada com base na % realizada*/
TUC *UCInsere(TUC *lista, TUC *elemento);
/* retorna a lista ordenada */
TUC *UCInsertSort(TUC *lista);
```

Claro que para as primeiras alíneas, não era necessário tudo isto, mas quem preveja fazer todas as alíneas pode fazer as primitivas básicas, adincionar e remover um elemento, e depois de verificar a necessidade de outras operações então faria essas primitivas, de forma independente do problema. Por exemplo a função para inverter as AFs, dado que ao serem carregadas ficam ao contrário, poderiam nessa altura identificar

que daria jeito inverter, pelo que basta fazer a função (ou replicar das AFs). Reparem que todas estas funções existem em atividades formativas. O algoritmo de ordenação poderia ser o MergeSort (houve quem tivesse implementado), mas como claramente o número de elementos é reduzido, não é da ordem dos milhares ou milhões, é desnecessário o merge sort, basta qualquer algoritmo, o insert sort é suficiente.

Claro que para a alínea A, uma função para carregar as UCs, e para a alínea B outra função para carregar as AFs, utilizando as primitivas, seria suficiente.

No resto do código não há malloc ou free. Apenas se utilizam as primitivas, se for detectada nova necessidade, cria-se outra primitiva que depende apenas do tipo abstrato de dados e válido para qualquer problema.

Houve problemas em ler uma sequência de números com um separador, poderiam ter utilizado o scanf, ou por exemplo fazer a seguinte função (já que seria reutilizada na alínea A e B):

```
char *LerNumero(int *valor, char *str) {
    if (str != NULL) {
        *valor = atoi(str);
        str = strstr(str, " - ");
        if (str != NULL)
            str += 3;
    }
    return str;
}
```

Esta função permitiria ler um número e obter o apontador para continuar a processar a string, por exemplo:

```
nome = LerNumero(&inicio, nome);
```

Esta instrução dá à função o local onde está a processar, carrega o número para a variável inicio, e retorna o novo apontador para continuar a processar o número seguinte.

Aqui está uma proposta para carregar as UCs e AFs:

```
TUC *CarregaUCs(TUC *lista, FILE *f)
{
    char str[MAXSTR];
    int letras, numeroUC;
    char *nome;
    while (fgets(str, MAXSTR, f) != NULL && (letras = strlen(str)) >
1) {
        if (str[letras - 1] == '\n')
            str[letras - 1] = 0;
        nome = LerNumero(&numeroUC, str);
        lista = UCAdiciona(lista, numeroUC, nome);
    }
    return lista;
}
int CarregaAtividades(TUC *lista, FILE *f)
{
    char str[MAXSTR];
```

```
    int letras, numeroUC, inicio, fim, realizado, sessoes, contagem=0;
    char *nome;
    while (fgets(str, MAXSTR, f) != NULL && (letras = strlen(str)) >
1) {
        if (str[letras - 1] == '\n')
            str[letras - 1] = 0;
        nome = LerNumero(&numeroUC, str);
        nome = LerNumero(&inicio, nome);
        nome = LerNumero(&fim, nome);
        nome = LerNumero(&realizado, nome);
        nome = LerNumero(&sessoes, nome);
        ActAdiciona(UCElemento(lista,numeroUC), nome,
            inicio, fim, realizado, sessoes);
        contagem++;
    }
    // inverter todas as atividades para ficar por ordem de inserção
    for (; lista != NULL; lista = lista->seg)
        lista->atividades = ActInverte(lista->atividades);
    return contagem;
}
```

Claro que este método está a contar todas as atividades formativas, mas algumas podem não ter sido colocadas dado que pertencem a UCs ineixistentes. É necessário fazer uma passagem pela estrutura de dados para contabilizar as atividades formativas inseridas corretamente. Por outro lado, na alínea C é necessário atualizar a percentagem da UC realizada, pelo que pode ficar também nesta função.

```
void ContagemAtividades(TUC *lista, int *atividades, int *realizadas)
{
    int totalSessoes; // alínea C, atualizar a percentagem da UC
realizada
    int sessoesRealizadas;
    TActividade *aux;
    while (lista != NULL) {
        aux = lista->atividades;
        totalSessoes = sessoesRealizadas = 0;
        while (aux != NULL) {
            totalSessoes += aux->sessoes;
            (*atividades)++;
            if (aux->realizada>0) {
                sessoesRealizadas += aux->sessoes;
                (*realizadas)++;
            }
            aux = aux->seg;
        }
        if (totalSessoes > 0)
            lista->realizada = 100.*sessoesRealizadas / totalSessoes;
        else
            lista->realizada = 0;
        lista = lista->seg;
    }
}
```

Como se pretende transformar uma percentagem em nota, não vale a pena estar a guardar dois registos, basta uma pequena função dado que tem dois segmentos de reta:

```
int NotaPrevista(float percentagem)
{
    if (percentagem >= 25.0)
```

```
        return 10 + 10 * (percentagem - 25.0) / 75.0 + 0.5;
    return 10 * percentagem / 25.0 + 0.5;
}
```

Houve muita confusão, e também muitos a utilizarem funções de arredondamento. Vale a pena memorizar o nome de uma função quando se sabe que a conversão de um número real para inteiro é por defeito? Se quisermos que o arredondamento seja para o mais próximo, basta somar 0.5.

Para a última alínea é que existia alguma complexidade, pelo que convinha que tivessem identificado a necessidade de guardar o número de AFs disponíveis em cada sessão de estudo (em vez de alguma eventual optimização por serem 2 o limite). Portanto convém aqui um vetor com um número por cada sessão de estudo. Primeiro é preciso saber quantas sessões de estudo existem:

```
int SessoesEstudo(TUC*lista)
{
    TActividade *listaAct;
    int ultima = -1;
    for( ; lista != NULL; lista = lista->seg)
        for (listaAct = lista->atividades; listaAct != NULL; listaAct
= listaAct->seg) {
            if (ultima== -1 || listaAct->fim > ultima)
                ultima = listaAct->fim;
        }
    return ultima;
}
```

De seguida, convém poder calendarizar uma atividade, após escolher a atividade, no primeiro instante em que tal é possível. Esta função tem de saber a atividade e o vetor com as AFs disponíveis:

```
void CalendarizarAct(TActividade *act, int *atividadesEmCurso)
{
    int i, j;
    for (i = act->inicio; i <= act->fim - act->sessoes+1; i++) {
        // verificar se há espaço
        for (j = i; j < i + act->sessoes; j++)
            if (atividadesEmCurso[j] <= 0)
                break;
        if (j == i + act->sessoes) { // existe espaço, subtrair
            for (j = i; j < i + act->sessoes; j++)
                atividadesEmCurso[j]--;
            act->realizada = i + 2;
            break;
        }
    }
    if (i > act->fim - act->sessoes + 1) { // não há hipótese para
esta atividade
//        printf("\n#%s: inicio %d fim %d sessoes %d",
//            act->nome, act->inicio, act->fim, act->sessoes);
        act->realizada = -1;
    }
}
```

Agora estamos em condições de fazer a função principal, calendarizar o estudo, que recebe o valor K do número de UCs a focar:

```
TUC *CalendarioEstudo(TUC *lista, int UCs)
{
    TActividade *act, *aux;
    int *atividadesEmCurso;
    int i, j, k, ultima, nsessoes, atividades, realizadas;
    ultima=SessoesEstudo(lista);
    // existem sessões de estudo de 0 a ultima-primeira+1
    // guardar para cada sessão 2 atividades, para cada atividade
    nsessoes = ultima + 1;
    atividadesEmCurso = (int*)malloc(sizeof(int)*nsessoes);
    if (atividadesEmCurso == NULL)
        return;
    for (j = 0; j < nsessoes; j++)
        atividadesEmCurso[j] = ACT_SESSAO;
    ContagemAtividades(lista, &atividades, &realizadas);
    lista = UCInsertSort(lista);
    for (k = UCs - 1; k >= 0; k--) {
        // localizar a atividade mais fácil na UC com nota mais baixa
        act = aux = UCkElemento(lista, k)->atividades;
        for (; aux != NULL; aux = aux->seg) {
            if (act->realizada!=0 ||
                aux->realizada==0 && (act->sessoes>aux->sessoes ||
                    act->sessoes==aux->sessoes &&
                    (act->fim-act->inicio)>(aux->fim - aux->inicio)))
                act = aux;
        }
        if (act->realizada == 0) {
            // processar esta atividade
            CalendarizarAct(act, atividadesEmCurso);
            if (act->realizada > 0) {
                ContagemAtividades(lista, &atividades, &realizadas);
                lista = UCInsertSort(lista);
                k = UCs; // reconsiderar em todas as UCs já que
realizou uma AF
            }
            else
                k++; // manter esta UC, retirou-se uma AF
        }
    }
    free(atividadesEmCurso);
    return lista;
}
```

É conveniente ter é claro uma função para mostrar o resultado, nunca misturando output com processamento. Apenas faz sentido colocar output em funções que processam os dados, se for para efeitos de debug. Caso contrário é uma excelente oportunidade separar o trabalho de processar, do de visualizar. Pode a especificação mudar apenas na visualização, então o processamento fica inalterado, ou vice-versa. E mesmo que não existisse esta situação, todas as oportunidades de dividir código, que façam sentido, devem aproveitar, dado que ficam com pedaços mais pequenos.

```
void PlanoEstudo(TUC *lista)
{
    int i;
    TActividade *aux;
    while (lista != NULL) {
        printf("\n%s: UC%d %.1f%% Nota prevista: %d valores",
            lista->nome, lista->numero, lista->realizada,
            NotaPrevista(lista->realizada));
```

```
        for (aux = lista->atividades; aux != NULL; aux = aux->seg)
            if (aux->realizada > 1) {
                printf("\n   %s, sessoes:", aux->nome);
                for (i = 0; i < aux->sessoes; i++)
                    printf(" %d", aux->realizada - 2 + i);
            }
        lista = lista->seg;
    }
}
```

Não coloquei anteriormente, mas para a alínea C teria também de existir uma função para mostrar o resultado, claro que mais simples:

```
void RelatorioUC(TUC*lista) {
    while (lista != NULL) {
        printf("\n%s: UC%d %.1f%% Nota prevista: %d valores",
            lista->nome, lista->numero, lista->realizada,
            NotaPrevista(lista->realizada));
        lista = lista->seg;
    }
}
```

Naturalmente que a função main, acaba por ficar simples, dado que toda a funcionalidade ficou tratada, teriam de lidar com o facto de existirem várias alíneas.

```
int main()
{
    char str[MAXSTR]="";
    FILE *f = stdin;
    TUC *lista, *aux;
    int tamanho = 0, atividades, realizadas, k;
    lista = CarregaUCs(NULL, f);
    atividades = CarregaAtividades(lista, f);
    fgets(str, MAXSTR, f);
    for (aux = lista; aux != NULL; aux = aux->seg)
        tamanho += strlen(aux->nome);
    printf("%d", tamanho);
    if (atividades > 0) {
        printf(" %d", atividades);
        atividades = 0;
        realizadas = 0;
        ContagemAtividades(lista, &atividades, &realizadas);
        printf(" %d %d", atividades, realizadas);
    }
    if (str[0] == '0') { // alínea C
        lista = UCInsertSort(lista);
        RelatorioUC(lista);
    }
    else if(strlen(str)>1) { // alínea D
        lista=CalendarioEstudo(lista, k = atoi(str));
        PlanoEstudo(lista);
    }
    // libertar a estrutura de dados
    while (lista != NULL)
        lista = UCRemove(lista);
}
```