1. Uma solucao para o EP2. Esta é uma implementação em `CWEB` de um algoritmo de escalonamento simples. O propósito é resolver o EP2. Nós nos baseamos firmemente no programa `ex11.w`, de ordenação topológica.

Observação (de sempre). A versão do `CWEB` que estamos usando neste programa não mescla muito bem com o português: um projeto interessante seria encontrar uma solução simples para este problema.

2. Aqui está a estrutura geral do nosso programa `ep2.w`.

```
⟨ Inclusão de arquivos sobre as bibliotecas 3 ⟩
⟨ Declaração de tipos 5 ⟩
⟨ Variáveis globais 4 ⟩
⟨ O programa principal 6 ⟩
```

3. Precisamos incluir os arquivos `.h` para as funções de entrada e saída e para as funções de alocação de memória.

```
⟨ Inclusão de arquivos sobre as bibliotecas 3 ⟩ ≡
#include <stdio.h>    /* para as funções de entrada e saída */
#include <stdlib.h>   /* para calloc( ), malloc( ), e free( ) */
```

This code is used in section 2.

4. Usaremos a variável *status* para passar ao sistema operacional informação sobre o sucesso, ou não, da execução deste programa. A variável *nome_do_prog* é usada para guardar o nome do programa (como em `wc.w`).

```
#define OK 0    /* valor de status quando tudo correu bem */
#define erro_nos_dados 1    /* valor de status quando os dados de entrada foram inconsistentes */
#define erro_na_leitura_de_arquivo 2
        /* valor de status para quando ocorre erro na leitura da entrada */
#define erro_na_alocacao_de_memoria 3
        /* valor de status para quando ocorre erro na alocação de memória */
#define erro_no_numero_de_rodadas 4    /* esperemos que isto nunca aconteca */
⟨ Variáveis globais 4 ⟩ ≡
  int status = OK;    /* inicialmente nada aconteceu de errado */
  char *nome_do_prog;    /* o nome deste programa */
  char *nome_do_arq;    /* nome do arquivo de dados, caso não seja o stdin */
  FILE *arquivo = Λ;    /* ponteiro para arquivo de dados (se diferente de stdin) */
```

This code is used in section 2.

5. Os elementos da ordem parcial serão representados por estruturas. Tais estruturas (do tipo **struct** *elemento*) terão os campos *id*, *g_entr*, *topo_suc*, *prox_fila*, e *anterior*. O campo *id* é simplesmente um inteiro que identifica o elemento. O campo *g_entr* é um inteiro, que contará o número de *arcos* que terminam neste elemento em cada instante.

Queremos montar uma partição dos elementos da entrada em *rodadas*. A primeira rodada é constituída dos elementos que têm $g\_entr = 0$, i.e., as *fontes*. Removemos então todos os elementos desta primeira rodada (e também os arcos que emanam deles). Tal remoção cria novas fontes, e a segunda rodada é constituída destas novas fontes. Prosseguimos analogamente até esgotarmos a entrada.

Como no algoritmo de ordenação topológica, manteremos, para cada x, uma lista de estruturas que representam os elementos para os quais x manda um arco. Estas estruturas serão do tipo **struct** *sucessor*. A lista correspondente ao elemento x começa na estrutura *x→topo_suc*. O campo *prox_suc* de uma estrutura do tipo **struct** *sucessor*, que é declarado como um ponteiro para **struct** *sucessor*, será usado para implementar esta lista. O campo *id* de uma estrutura do tipo **struct** *sucessor* aponta para o elemento da ordem parcial que ele representa (mais precisamente, para a estrutura do tipo **struct** *elemento* que representa o elemento).

O campo *prox_fila* dos **struct** *elemento* será usado para implementar uma fila que conterá as fontes, i.e., os elementos com $g\_entr = 0$. O campo *anterior* destas estruturas, que será um ponteiro para **struct** *elemento*, será usado para se construir seqüências de elementos que formam "cadeias de dependência completa" (os elementos de cada uma destas seqüências dependem um do outro, em uma ordem linear). Se o nosso programa devolve uma decomposição da entrada em k rodadas, então devemos também devolver uma tal seqüência com k elementos.

Montaremos uma lista de listas para representar a solução final. A k-ésima lista desta lista conterá os elementos que constituem a k-ésima rodada. Usaremos o campo *prox_na_rodada* para implementar as listas que representam as rodadas. Aqui vamos tentar ser econômicos, e definiremos *prox_na_rodada* sendo uma macro que vale *prox_fila* (lembre-se da observação sobre "leitores astutos" de `ex11.w`).

Finalmente, usaremos as estruturas **struct** *lista* para montar a lista das listas das rodadas.

```
#define prox_na_rodada prox_fila

⟨ Declaração de tipos 5 ⟩ ≡
  struct elemento {
    int id;
    int g_entr;
    struct sucessor *topo_suc;
    struct elemento *prox_fila;
    struct elemento *anterior;
  };
  struct sucessor {
    struct elemento *id;
    struct sucessor *prox_suc;
  };
  struct lista {
    struct elemento *comeco;
    struct lista *prox_lista;
  };
```

This code is used in section 2.

6. A função *main* tem a seguinte cara.

```
⟨ O programa principal 6 ⟩ ≡
  main(argc, argv)
      int argc;    /* o número argumentos na linha de comando */
      char **argv;    /* um vetor de strings, contendo os argumentos */
  { ⟨ Variáveis locais a main 12 ⟩
    nome_do_prog = argv[0];
    ⟨ Interprete a linha de comando 7 ⟩;
    ⟨ Leia os dados de entrada 8 ⟩;
    ⟨ Encontre uma decomposição em rodadas 13 ⟩;
    ⟨ Imprima a decomposição encontrada 23 ⟩;
    ⟨ Imprima uma seqüência de elementos interdependentes 25 ⟩;
    exit(status);
  }
```

This code is used in section 2.

7. O usuário pode fornecer os dados de entrada de duas formas. A mais simples é através do *stdin*. Entretanto, podemos também fornecer um nome de arquivo na linha de comando (o primeiro argumento fornecido será interpretado como o nome do arquivo).

```
#define READ_ONLY "r"

⟨ Interprete a linha de comando 7 ⟩ ≡
  if (argc > 1) {    /* o usuário forneceu pelo menos um argumento */
    nome_do_arq = argv[1];
    arquivo = fopen(nome_do_arq, READ_ONLY);
    if (arquivo ≡ Λ) {
      status = erro_na_leitura_de_arquivo;
      fprintf(stderr, "%s:␣Erro␣na␣abertura␣do␣arquivo␣%s\n", nome_do_prog, nome_do_arq);
      exit(status);
    }
  }    /* caso contrário, a entrada virá no stdin. Neste caso, arquivo = Λ */
```

This code is used in section 6.

8. Agora que já sabemos de onde virá a entrada, precisamos lê-la. Assumimos que o número de elementos n da ordem parcial vem na primeira linha do arquivo de entrada. Nas linhas seguintes, assumimos que vêm os pares (i, j), um par por linha; o par (i, j) representa a relação $i \prec j$. No arquivo, assumimos que o par (i, j) é representado por dois inteiros separados por espaços.

```
⟨ Leia os dados de entrada 8 ⟩ ≡
  { int scanf_args;    /* o número de argumentos de scanf que foram entendidos com sucesso */
    if (arquivo) scanf_args = fscanf(arquivo, "%d", &n);
    else scanf_args = scanf("%d", &n);
    if (scanf_args < 1) {
      status = erro_nos_dados;
      fprintf(stderr, "%s:␣Erro␣na␣leitura␣do␣numero␣de␣elementos.\n", nome_do_prog);
      exit(status);
    }
    ⟨ Aloque e inicialize memória para as n estruturas 9 ⟩;
    ⟨ Leia as relações fornecidas 10 ⟩;
    if (arquivo) fclose(arquivo);
  }
```

This code is used in section 6.

9. Fazemos aqui a alocação de memória para as n estruturas que vão representar os elementos da ordem parcial. Uma outra opção seria o uso de um vetor de `NMAX` entradas, com algum `NMAX` grande.

⟨ Aloque e inicialize memória para as n estruturas 9 ⟩ ≡

```
{ int k;
  struct elemento *ptr;
  els = (struct elemento *) calloc(n, sizeof(struct elemento));
  if (els ≡ Λ) {    /* faltou memória... */
    status = erro_na_alocacao_de_memoria;
    fprintf(stderr, "%s:␣Faltou␣memoria!\n", nome_do_prog);
    exit(status);
  }
  for (ptr = els, k = 1; ptr < els + n; ptr++, k++) {
    ptr→id = k;
    ptr→g_entr = 0;
    ptr→topo_suc = Λ;
    ptr→prox_fila = ptr→anterior = Λ;
  }
}
```

This code is used in section 8.

10. Agora precisamos ler os vários pares (i, j) a serem lidos. Para tanto, lemos dois inteiros por vez, e quando não conseguimos ler um par de inteiros assumimos que o arquivo terminou.

⟨ Leia as relações fornecidas 10 ⟩ ≡

```
{ int i, j;    /* os elementos lidos */
  int scanf_args;    /* o número de argumentos de scanf que foram entendidos com sucesso */
  int ha_pares_para_ler = 1;    /* variável booleana para controle da leitura dos pares */
  while (ha_pares_para_ler) {
    if (arquivo) scanf_args = fscanf(arquivo, "%d␣%d", &i, &j);
    else scanf_args = scanf("%d␣%d", &i, &j);
    if (scanf_args < 2) ha_pares_para_ler = 0;
    else ⟨ Insira o par (i, j) em nossa estrutura de dados 11 ⟩;
  }
}
```

This code is used in section 8.

11. Queremos agora registrar em nossa estrutura de dados que temos $i \prec j$.

#define $el(k)$ $(els + k - 1)$

⟨ Insira o par (i, j) em nossa estrutura de dados 11 ⟩ ≡

```
{ struct sucessor *ptr;
  ptr = (struct sucessor *) malloc(sizeof(struct sucessor));
  if (ptr ≡ Λ) {      /* faltou memória... */
    status = erro_na_alocacao_de_memoria;
    fprintf(stderr, "%s:␣Faltou␣memoria!\n", nome_do_prog);
    exit(status);
  }
  ptr→id = el(j);
  el(j)→g_entr++;      /* incrementamos o contador de arcos de entrada para j */
  ptr→prox_suc = el(i)→topo_suc;
  el(i)→topo_suc = ptr;      /* inserimos o elemento j na lista de sucessores de i */
}
```

This code is used in section 10.

12. Comecemos agora com o algoritmos propriamente dito. No que segue, chamamos de *fontes* os elementos que já não recebem nenhum arco ou, para ser mais preciso, aqueles elementos cujos "ancestrais" foram todos já classificados em alguma rodada anterior. Manteremos o conjunto de fontes em uma fila. O começo da fila será apontada por *comeco* e o término por *fim*.

⟨ Variáveis locais a *main* 12 ⟩ ≡

```
struct elemento *comeco, *fim;
```

See also sections 14, 20, and 26.

This code is used in section 6.

13. O algoritmo em si está nesta parágrafo.

⟨ Encontre uma decomposição em rodadas 13 ⟩ ≡

```
⟨ Inicialize a lista das listas das rodadas 15 ⟩;
⟨ Inicialize a fila de fontes 22 ⟩;
⟨ Monte as listas que representam as rodadas 16 ⟩;
⟨ Verifique se classificamos todos os elementos 21 ⟩;
```

This code is used in section 6.

14. A lista das listas das rodadas será na verdade uma fila, pois queremos que as rodadas sejam acrescentadas no fim da lista, para que a ordem das rodadas seja "crescente." O início e fim desta fila serão apontados por *primeira_rodada* e *rodada_mais_recente*. Manteremos também um contador *n_rodadas* com o número de rodadas que já temos.

⟨ Variáveis locais a *main* 12 ⟩ +≡

```
struct lista *primeira_rodada;
struct lista *rodada_mais_recente;
int n_rodadas = 0;
```

15. ⟨ Inicialize a lista das listas das rodadas 15 ⟩ ≡

```
primeira_rodada = rodada_mais_recente = Λ;
```

This code is used in section 13.

16. ⟨ Monte as listas que representam as rodadas 16 ⟩ ≡

```
while (comeco) {      /* enquanto a fila das fontes não estiver vazia... */
  n_rodadas++;
  ⟨ Coloque todos os elementos da fila de fontes na fila das rodadas 17 ⟩;
  ⟨ Percorra a lista desta nova rodada e acerte os g_entr e a fila de fontes 18 ⟩;
}
```

This code is used in section 13.

17. ⟨ Coloque todos os elementos da fila de fontes na fila das rodadas 17 ⟩ ≡

```
{ struct lista *ptr;
  ptr = (struct lista *) malloc(sizeof(struct lista));
  ptr→comeco = comeco;
  ptr→prox_lista = Λ;
  if (rodada_mais_recente ≡ Λ) primeira_rodada = rodada_mais_recente = ptr;
  else {
    rodada_mais_recente→prox_lista = ptr;
    rodada_mais_recente = ptr;
  }
  comeco = fim = Λ;      /* a fila de fontes está vazia agora */
}
```

This code is used in section 16.

18. ⟨ Percorra a lista desta nova rodada e acerte os g_entr e a fila de fontes 18 ⟩ ≡

```
{ struct elemento *ptr_elemento;
  struct sucessor *ptr_suc, *ptr_aux;
  ptr_elemento = rodada_mais_recente→comeco;
  while (ptr_elemento) {
    ptr_suc = ptr_elemento→topo_suc;
    while (ptr_suc) {
      if (−−ptr_suc→id→g_entr ≡ 0) ⟨ Processe a nova fonte ptr_suc→id 19 ⟩;
      ptr_aux = ptr_suc;
      ptr_suc = ptr_suc→prox_suc;
      free(ptr_aux);
    }
    n_proc++;
    ptr_elemento = ptr_elemento→prox_na_rodada;
  }
}
```

This code is used in section 16.

19. Quando descobrimos que um elemento x se tornou uma fonte, devemos inseri-lo na fila das fontes e também acertar *x→anterior*, que passa a apontar para o elemento que mandava para x o último arco que ele recebia.

```
⟨ Processe a nova fonte ptr_suc→id 19 ⟩ ≡
  { ptr_suc→id→prox_fila = Λ;
    if (fim ≡ Λ) comeco = fim = ptr_suc→id;
    else {
      fim→prox_fila = ptr_suc→id;
      fim = ptr_suc→id;
    }
    ptr_suc→id→anterior = ptr_elemento;
  }
```

This code is used in section 18.

20. Queremos verificar, ao término da execução do algoritmo, se classificamos todos os n elementos em rodadas. Para tanto, mantemos um contador que conta o número de elementos que são classificados em rodadas.

```
⟨ Variáveis locais a main 12 ⟩ +≡
  int n_proc = 0;
```

21.

```
⟨ Verifique se classificamos todos os elementos 21 ⟩ ≡
  if (n_proc < n) {
    fprintf(stderr, "%s:␣Erro␣nos␣dados␣de␣entrada;␣ha␣ciclos.\n", nome_do_prog);
    printf("Vai␣aqui␣a␣solucao␣parcial␣atual...\n");
    ⟨ Imprima as rodadas 24 ⟩;
    status = erro_nos_dados;
    exit(status);
  }
```

This code is used in section 13.

22. Antes de começar o algoritmo, devemos encontrar e inserir todas as fontes na lista de fontes.

```
⟨ Inicialize a fila de fontes 22 ⟩ ≡
  { int k;

    comeco = fim = Λ;
    for (k = 1; k ≤ n; k++)
      if (el(k)→g_entr ≡ 0)
        if (comeco ≡ Λ)     /* fila vazia */
          fim = comeco = el(k);
        else {     /* fila não está vazia */
          fim→prox_fila = el(k);
          fim = el(k);
        }
  }
```

This code is used in section 13.

23.

```
⟨ Imprima a decomposição encontrada 23 ⟩ ≡
  printf("Uma␣decomposicao␣em␣rodadas␣da␣ordem␣parcial␣e␣como␣segue:\n");
  ⟨ Imprima as rodadas 24 ⟩;
```

This code is used in section 6.

24. ⟨ Imprima as rodadas 24 ⟩ ≡

```
{ struct lista *ptr_lista;
  struct elemento *ptr_elemento;
  int k;
  for (k = 1, ptr_lista = primeira_rodada; ptr_lista; k++, ptr_lista = ptr_lista→prox_lista) {
    printf("%d:␣", k);
    for (ptr_elemento = ptr_lista→comeco; ptr_elemento; ptr_elemento = ptr_elemento→prox_na_rodada)
      printf("%d␣", ptr_elemento→id);
    printf("\n");
  }
}
```

This code is used in sections 21 and 23.

25. Se listamos a seqüência de elementos que começa com um elemento que pertence à ultima rodada, digamos x, seguida de *x→anterior*, seguida de *x→anterior→anterior* etc, encontraremos uma seqüência de elementos que demonstram que o número de rodadas em que decompusemos a entrada é ótima, i.e., não poderíamos ter decomposto em menos rodadas. Um pequeno detalhe é que, na verdade, fica mais intuitivo imprimirmos uma seqüência com descrita acima na ordem reversa.

⟨ Imprima uma seqüência de elementos interdependentes 25 ⟩ ≡

```
{ struct elemento *ptr_elemento;
  struct elemento *topo = Λ;
  struct elemento *ptr_aux = Λ;
  int k = 0;
  ptr_elemento = rodada_mais_recente→comeco;
  while (ptr_elemento) {
    ptr_aux = ptr_elemento;
    ptr_elemento = ptr_elemento→anterior;
    ptr_aux→anterior = topo;
    topo = ptr_aux;
    k++;
  }
  printf("Vai␣aqui␣o␣caminho␣que␣encontrei␣(%d␣elementos):\n", k);
  for ( ; topo; topo = topo→anterior) printf("%d␣", topo→id);
  printf("\n");
  if (k ≠ n_rodadas) {
    status = erro_no_numero_de_rodadas;
    fprintf(stderr, "%s:␣Erro␣no␣numero␣de␣rodadas␣x␣comprimento␣do␣caminho␣encontrado!\n",
        nome_do_prog);
    exit(status);
  }
}
```

This code is used in section 6.

26. ⟨ Variáveis locais a *main* 12 ⟩ +≡

```
int n;    /* número de elementos na ordem parcial */
struct elemento *els;    /* ponteiro para os elementos */
```

27. Indice. Aqui está uma lista de identificadores que são usados, com indicação de onde eles aparecem. As entradas sublinhadas indicam onde eles aparecem.

⟨ Aloque e inicialize memória para as n estruturas 9 ⟩ Used in section 8.
⟨ Coloque todos os elementos da fila de fontes na fila das rodadas 17 ⟩ Used in section 16.
⟨ Declaração de tipos 5 ⟩ Used in section 2.
⟨ Encontre uma decomposição em rodadas 13 ⟩ Used in section 6.
⟨ Imprima a decomposição encontrada 23 ⟩ Used in section 6.
⟨ Imprima as rodadas 24 ⟩ Used in sections 21 and 23.
⟨ Imprima uma seqüência de elementos interdependentes 25 ⟩ Used in section 6.
⟨ Inclusão de arquivos sobre as bibliotecas 3 ⟩ Used in section 2.
⟨ Inicialize a fila de fontes 22 ⟩ Used in section 13.
⟨ Inicialize a lista das listas das rodadas 15 ⟩ Used in section 13.
⟨ Insira o par (i, j) em nossa estrutura de dados 11 ⟩ Used in section 10.
⟨ Interprete a linha de comando 7 ⟩ Used in section 6.
⟨ Leia as relações fornecidas 10 ⟩ Used in section 8.
⟨ Leia os dados de entrada 8 ⟩ Used in section 6.
⟨ Monte as listas que representam as rodadas 16 ⟩ Used in section 13.
⟨ O programa principal 6 ⟩ Used in section 2.
⟨ Percorra a lista desta nova rodada e acerte os *g_entr* e a fila de fontes 18 ⟩ Used in section 16.
⟨ Processe a nova fonte *ptr_suc→id* 19 ⟩ Used in section 18.
⟨ Variáveis globais 4 ⟩ Used in section 2.
⟨ Variáveis locais a *main* 12, 14, 20, 26 ⟩ Used in section 6.
⟨ Verifique se classificamos todos os elementos 21 ⟩ Used in section 13.