1. Um programa para ordenacao topologica. Esta é uma implementação em CWEB do algoritmo de ordenação topológica vista em aula. Embora não conheçamos CWEB em detalhe, a leitura de uma programa nesta linguagem não deve ser difícil, já que o código-fonte é impresso de forma bastante intuitiva. Para maiores detalhes sobre esta linguagem, a favorita de D. E. Knuth, veja, por exemplo, *The* CWEB *System of Structured Documentation, de Donald E. Knuth and Sílvio Levy (Reading, Massachusetts: Addison-Wesley, 1993), iv+227pp. ISBN 0-201-57569-8.* O leitor atento notará que este programa foi baseado, em sua organização geral, no programa `wc.w` de Knuth e Levy, discutido em sala.

Observação. 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 `ex11.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 */
```

⟨ 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_ord*, e *prox_fila*. 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.

Lembre que os elementos são inclusos um por um na ordenação linear L que devolveremos na saída. Quando um elemento, digamos x, é inserido em L, todos os arcos que emanam deste elemento são desconsiderados. Note que, desta forma, é interessante termos uma forma fácil de saber quais são os elementos y para os quais x manda um arco.

Para podermos identificar estes elementos y (os "sucessores imediatos" de x), manteremos, para cada x, uma lista de estruturas que representam estes elementos. 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).

Lembre-se ainda de que os elementos que podem ser inclusos na ordenação linear L em um dado momento são exatamente aqueles que não recebem nenhum arco, isto é, aqueles para os quais $g\_entr = 0$. O campo *prox_fila* será usado para implementar uma fila que conterá os elementos com $g\_entr = 0$.

A solução final de nosso algoritmo, uma ordenação linear dos elementos da ordem parcial de entrada, será representada como uma lista ligada, e usaremos o campo *prox_ord*, que é declarado como um ponteiro para elementos, para implemetar esta lista. O leitor astuto perceberá que poderíamos usar o campo *prox_fila* para este fim, mas provavelmente a economia de espaço não compensa (e ganhamos clareza mantendo dois campos diferentes para dois propósitos diferentes).

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

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 ordenação topológica 13 ⟩;
    ⟨ Imprima a ordenação encontrada 17 ⟩;
    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_ord = ptr→prox_fila = 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 postos na ordenação linear de saída. 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 section 19.

This code is used in section 6.

13. O algoritmo em si está nesta parágrafo.

⟨ Encontre uma ordenação topológica 13 ⟩ ≡

```
⟨ Inicialize a fila de fontes 16 ⟩;
⟨ Monte a lista de saída 14 ⟩;
⟨ Verifique se ordenamos todos os elementos 15 ⟩;
```

This code is used in section 6.

14. ⟨ Monte a lista de saída 14 ⟩ ≡

```
while (comeco) {    /* enquanto a fila das fontes não estiver vazia... */
  struct sucessor *ptr, *tmp;
  ptr = comeco→topo_suc;
  while (ptr) {
    ptr→id→g_entr−−;
    if (ptr→id→g_entr ≡ 0) {    /* ptr→id se tornou uma nova fonte */
      ptr→id→prox_fila = Λ;
      fim→prox_fila = ptr→id;
      fim = ptr→id;
    }    /* inserimos no fim da fila das fontes esta nova fonte */
    tmp = ptr;
    ptr = ptr→prox_suc;
    free(tmp);    /* acabamos de eliminar o arco (i, j), onde o elemento i está no começo da fila e j é o
        elemento representado por ptr. */
  }    /* acabamos de processar todos os sucessores do elemento i que estava no começo da fila de
        fontes. Este processo equivale a "eliminar" o elemento i (e todos os arcos que emanam dele) da
        nossa ordem parcial. */
  comeco→prox_ord = sol;
  sol = comeco;    /* agora incluímos o elemento no comeco da fila para a lista solucao */
  n_proc++;    /* mais um elemento processado */
  comeco = comeco→prox_fila;    /* agora eliminamos o primeiro elemento da fila */
}
```

This code is used in section 13.

15. ⟨ Verifique se ordenamos todos os elementos 15 ⟩ ≡

```
if (n_proc < n) {
  fprintf(stderr, "%s: Erro nos dados de entrada; ha ciclos.\n", nome_do_prog);
  printf("Vai aqui a lista sol atual...\n");
  ⟨ Imprima o conteúdo da lista sol na ordem reversa 18 ⟩;
  status = erro_nos_dados;
  exit(status);
}
```

This code is used in section 13.

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

⟨ Inicialize a fila de fontes 16 ⟩ ≡

```
{ 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.

17. Pequena rotina de impressão da lista ligada cujo começo é apontado por *sol*. Uma sutileza é que devemos inverter a lista antes de imprimi-la.

⟨ Imprima a ordenação encontrada 17 ⟩ ≡

```
printf("Uma␣ordenacao␣topologica␣desta␣ordem␣parcial␣e␣como␣segue:\n");
⟨ Imprima o conteúdo da lista sol na ordem reversa 18 ⟩;
```

This code is used in section 6.

18. Note que para imprimirmos os elementos na ordem "crescente" (de forma que os arcos "vão da esquerda para a direita"), devemos de fato imprimir os elementos na ordem reversa àquela da lista *sol*.

⟨ Imprima o conteúdo da lista *sol* na ordem reversa 18 ⟩ ≡

```
{ struct elemento *ptr, *lista_final;
  lista_final = Λ;
  while (sol) {
    ptr = sol;
    sol = sol→prox_ord;
    ptr→prox_ord = lista_final;
    lista_final = ptr;
  }
  for (ptr = lista_final; ptr; ptr = ptr→prox_ord) printf("%d␣", ptr→id);
  printf("\n");
}
```

This code is used in sections 15 and 17.

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

```
int n;  /* número de elementos na ordem parcial */
int n_proc = 0;  /* número de elementos já processados (isto é, na ordenação linear de saída) */
struct elemento *els;  /* ponteiro para os elementos */
struct elemento *sol = Λ;  /* ponteiro para a lista que representa a ordenação solução */
```

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