1. Introdução. Esta é uma adaptação de um código escrito por Knuth [D. E. Knuth, *The Stanford GraphBase: a Platform for Combinatorial Computing*, (New York: ACM Press, 1993), viii+576pp. Co-publicado pela Addison–Wesley Publishing Company, ISBN 0-201-54275-7; Veja o módulo `GB_SORT` daquele pacote]. De fato, muita coisa é só tradução (e omissão de coisas além do escopo ou irrelevantes para nós).

```
#include <stdio.h>    /* o ponteiro NULL (Λ) é definido aqui */
  ⟨ Declarations 2 ⟩
  ⟨ The gb_linksort routine 5 ⟩
```

2. A rotina de maior interesse aqui é uma rotina chamada *gb_linksort*( ) que ordena (reorganiza) os elementos de uma lista ligada em ordem decrescente de seus campos *key*, que é um **long**. Assumimos que estes campos *key* contém inteiros não negativos menores que 2^{31}.

Após a ordenação, os elementos da lista original estarão organizados em 128 listas ligadas: *gb_sorted*[127], *gb_sorted*[126], ..., *gb_sorted*[0]. Para examinar os elementos em ordem decrescente de suas chaves, podemos varrer os elementos com rotinas como a seguinte:

```
{
  int j;
  node *p;
  for (j = 127; j ≥ 0; j−−)
    for (p = gb_sorted[j]; p; p = p→link)
      look_at(p);
}
```

Faremos isto abaixo para imprimir as chaves dos elementos em ordem decrescente. Naturalmente, não seria difícil reverter a ordem das listas.

Todos os elementos cujas chaves estão no intervalo $j2^{24} \le key < (j+1)2^{24}$ vão aparecer na lista *gb_sorted*[j]. Portanto, o resultado pode ser encontrado inteiramente contido na lista *gb_sorted*[0] se todas as chaves são estritamente menores que 2^{24}.

```
⟨ Declarations 2 ⟩ ≡
  typedef struct node_struct {
    long key;    /* quantidade numérica, que deve ser não-negativa */
    struct node_struct *link;    /* apontador para o próximo elemento da lista */
  } node;    /* estes serão os elementos da lista a ser processada */
```

See also section 4.

This code is used in section 1.

3. Aqui está o header file que os usuários deste módulo devem usar.

```
⟨ radix.h  3 ⟩ ≡
  typedef struct node_struct {
    long key;    /* quantidade numérica, que deve ser não-negativa */
    struct node_struct *link;    /* apontador para o próximo elemento da lista */
  } node;    /* estes serão os elementos da lista a ser processada */
  extern void gb_linksort( );    /* procedure to sort a linked list */
  extern node *gb_sorted[ ];    /* the results of gb_linksort */
```

4. Faremos 4 passadas do radix sort, com *radix* (base) 256. (Veja, por exemplo, Algoritmo 5.2.5R in *Sorting and Searching*, de Knuth). Nós movemos os elementos da lista entre dois vetores de listas: os vetores *gb_sorted* (que conterá o resultado final) e *alt_sorted* (que conterá resultados intermediários).

```
⟨ Declarations 2 ⟩ +≡
  node *gb_sorted[256];    /* vetor de listas, para as passadas pares */
  static node *alt_sorted[256];    /* vetor auxiliar, para as passadas ímpares */
```

5. Aqui estão as 4 passadas.

```
⟨ The gb_linksort routine 5 ⟩ ≡
  void gb_linksort(l)
      node *l;
  { register long k;      /* índice para a lista-destino */
    register node **pp;      /* ponto corrente na lista de ponteiros */
    register node *p, *q;      /* ponteiros para manipulação de listas */
    ⟨ Partition the given list into 256 sublists alt_sorted by low-order byte 6 ⟩;
    ⟨ Partition the alt_sorted lists into gb_sorted by second-lowest byte 7 ⟩;
    ⟨ Partition the gb_sorted lists into alt_sorted by second-highest byte 8 ⟩;
    ⟨ Partition the alt_sorted lists into gb_sorted by high-order byte 9 ⟩;
  }
```

This code is used in section 1.

6. ⟨ Partition the given list into 256 sublists *alt_sorted* by low-order byte 6 ⟩ ≡

```
  for (pp = alt_sorted + 255; pp ≥ alt_sorted; pp−−) *pp = Λ;
      /* inicializamos todas as listas-destino */
  for (p = l; p; p = q) {
    k = p→key & #ff;      /* extraímos os oito bits menos significantivos */
    q = p→link;
    p→link = alt_sorted[k];
    alt_sorted[k] = p;
  }
```

This code is used in section 5.

7. Agora vamos ler as listas de *alt_sorted* de 0 a 255, e vamos classificar os elementos de acordo com o segundo byte menos significativo.

```
⟨ Partition the alt_sorted lists into gb_sorted by second-lowest byte 7 ⟩ ≡
  for (pp = gb_sorted + 255; pp ≥ gb_sorted; pp−−) *pp = Λ;
      /* inicializamos todas as listas-destino */
  for (pp = alt_sorted; pp < alt_sorted + 256; pp++)
    for (p = *pp; p; p = q) {
      k = (p→key ≫ 8) & #ff;      /* extraímos os proximos oito bits */
      q = p→link;
      p→link = gb_sorted[k];
      gb_sorted[k] = p;
    }
```

This code is used in section 5.

8. Agora lemos as listas de 255 a 0, para obter a ordem correta no final. (Cada passada reverte a ordem dos elementos da lista; é meio peculiar mas funciona assim!)

```
⟨Partition the gb_sorted lists into alt_sorted by second-highest byte 8⟩ ≡
  for (pp = alt_sorted + 255; pp ≥ alt_sorted; pp−−) ∗pp = Λ;
      /∗ inicializamos todas as listas-destino ∗/
  for (pp = gb_sorted + 255; pp ≥ gb_sorted; pp−−)
    for (p = ∗pp; p; p = q) {
      k = (p→key ≫ 16) & #ff;      /∗ extraímos os proximos oito bits ∗/
      q = p→link;
      p→link = alt_sorted[k];
      alt_sorted[k] = p;
    }
```

This code is used in section 5.

9. Os 8 bits mais significativos corresponderão a números entre 0 e 127, porque assumimos que os campos *key* são inteiros menores que 2^{31}. (Uma rotina similar funcionaria para **int**, ou **unsigned long**, mas aí o funcionamento da rotina dependeria da representação binária dos números na máquina, o que afetaria a portabilidade deste programa.)

```
⟨Partition the alt_sorted lists into gb_sorted by high-order byte 9⟩ ≡
  for (pp = gb_sorted + 255; pp ≥ gb_sorted; pp−−) ∗pp = Λ;
      /∗ inicializamos todas as listas-destino ∗/
  for (pp = alt_sorted; pp < alt_sorted + 256; pp++)
    for (p = ∗pp; p; p = q) {
      k = (p→key ≫ 24) & #ff;      /∗ extraímos os oito bits mais significativos ∗/
      q = p→link;
      p→link = gb_sorted[k];
      gb_sorted[k] = p;
    }
```

This code is used in section 5.

10. Aqui está um programa muito simples para ilustrar o uso da rotina *gb_linksort*( ) implementada acima. Supomos que o usuário executa este programa com um parâmetro N; este programa então gera N inteiros aleatórios usando a função *rand*( ) e os coloca em uma lista ligada. Esta lista ligada é então fornecida como argumento em uma chamada de *gb_linksort*( ). A saída do programa é a lista de números gerados em ordem decrescente.

```
⟨driver.c 10⟩ ≡
#include <stdio.h>
#include <stdlib.h>
#include "radix.h"
  int main(int argc, char *argv[])
  {
    long j, k, N = atol(argv[1]);
    node *p, *q;
    p = Λ;
    while (N-- > 0) {
      q = (node *) malloc(sizeof(node));
      q→key = rand();
      q→link = p;
      p = q;
    }
    gb_linksort(p);
    for (j = 127; j ≥ 0; j--) {
      for (p = gb_sorted[j]; p; p = p→link) printf("%ld\n", p→key);
    }
    return 0;
  }
```

11. Índice. Aqui estão os identificadores usados neste programa, com seus respectivos locais de definição e uso.

RADIX_SORT