Курс «Алгоритмы и алгоритмические языки»
1 семестр 2017/2018

Лекция 23

Алгоритмы перебора множеств

- Перестановка некоторого набора элементов – это упорядоченная последовательность из этих элементов. Например, множество {1, 2, 3} имеет 6 различных перестановок (1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1).
- Для любого множества из n элементов существует ровно $n! = 1 \cdot 2 \cdot \ldots \cdot n$ различных перестановок.
- ***Задача*** состоит в том, чтобы написать программу, которая выводит все перестановки множества {1,2,..., n} в лексикографическом порядке (перестановки можно рассматривать как слова в алфавите B = {1, 2, ..., n})

Рекурсивный алгоритм генерации перестановок

- Будем перебирать перестановки чисел {1, 2, ..., *n*} в массиве *a*[*n*], последовательно заполняя его числами 1, ..., *n* в различном порядке.

  Для перебора всех перестановок будем записывать на первое место в массиве *a* по очереди числа 1, ..., *n*, и для каждого числа рекурсивно вызывать функцию генерации перестановок оставшихся *n* – 1 чисел. Массив *b* отмечает уже выбранные числа.

```
#include <stdio.h>
#include <stdlib.h>

void PrintPerm (int *a, int n) {
  int i;
  for (i = 0; i < n; i++)
    printf("%3d", a[i]);
  printf("\n");
}
```

Рекурсивный алгоритм генерации перестановок

```
void GenPerm (int *a, int *b, int i, int n) {
  if (i == n)
    PrintPerm (a, n);
  } else {
    int j;
    for (j = 0; j < n; j++)
      if (b[j] == 0) {
        b[j] = 1;
        a[i] = j + 1;
        GenPerm (a, b, i+1, n);
        b[j]  = 0;
      }
  }
}
```

Рекурсивный алгоритм генерации перестановок

```
int main (void) {
  int *a, *b;
  int n;
  scanf("%d", &n);
  a = malloc (n * sizeof (int));
  b = calloc (n, sizeof (int));
  GenPerm (a, b, 0, n); /* первый вызов генератора */
  free (a);
  free (b);
  return 0;
}
```

Нерекурсивный алгоритм генерации перестановок

- Задачу посещения (перебора) всех перестановок заданного множества можно свести к следующей задаче: по данной перестановке сгенерировать следующую за ней перестановку (например, в лексикографическом порядке). Один из первых алгоритмов решения этой задачи придумал индийский математик Пандит Нарайана еще в XIV веке.
- Алгоритм Нарайаны по любой данной перестановке из n элементов $a_1\ a_2\ ...\ a_n$ генерирует следующую (в лексикографическом порядке) перестановку.

  Если алгоритм Нарайаны применить в цикле к исходной последовательности n элементов $a_1\ a_2\ ...\ a_n$, отсортированных так, что $a_1 \le a_2 \le ... \le a_n$, то он сгенерирует все перестановки элементов множества $\{a_1\ a_2\ ...\ a_n\}$, в лексикографическом порядке.

Алгоритм Нарайаны

- Шаг 1. [Первая перестановка.] Составить перестановку $a_1\ a_2\ ...\ a_n$ ($a_1 < a_2 < ... < a_n$).

  Шаг 2. [Найти j, для которого $a_j < a_{j+1}$] Установить $j \leftarrow n - 1$. Если $a_j > a_{j+1}$, уменьшать j на 1 повторно, пока не выполнится условие $a_j < a_{j+1}$. Если окажется, что $j = 0$, завершить алгоритм.

  - На шаге 2 j является наименьшим индексом, для которого были посещены все перестановки, начиная с $a_1\ ...\ a_j$. Следовательно, лексикографически следующая перестановка увеличит значение a_j.

  Шаг 3. [Увеличить a_j] Установить $l \leftarrow n$. Если $a_j > a_l$, уменьшать l на 1, пока не выполняется условие $a_j < a_l$. Затем поменять местами $a_j \leftrightarrow a_l$.

  - Поскольку $a_{j+1} > ... > a_n$, элемент a_l является наименьшим элементом, который больше a_j, и который может следовать за $a_1\ ...\ a_{j-1}$ в перестановке. Перед заменой выполнялись отношения $a_{j+1} > ... > a_{l-1} > a_l > \mathbf{a_j} > a_{l+1} > ... > a_n$, а после замены – выполняются $a_{j+1} > ... > a_{l-1} > \mathbf{a_j} > a_l > a_{l+1} > ... > a_n$

  Шаг 4. [Обратить $a_{j+1}\ ...\ a_n$] Установить $k \leftarrow j + 1$ и $l \leftarrow n$. Затем, если $k < l$, поменять местами $a_k \leftrightarrow a_l$, установить $k \leftarrow k + 1$, $l \leftarrow l - 1$ и повторять, пока не выполнится условие $k > l$. Вернуться к шагу 2.

Алгоритм Нарайаны

```
#include <stdio.h>
#include <stdlib.h>

int NextPerm (int *a, int n) {
  int i, k, t, tmp;
  /* находим k такое что: a[k] < a[k+1] > .... > a[n-1]*/
  for (k = n - 2; (a[k] > a[k + 1]) && (k >= 0); k--);
  /* последняя перестановка */
  if (k == -1)
    return 0;
  /* находим t > k такое, что среди a[k+1],..., a[n-1]
     a[t] - минимальное число, большее a[k] */
  for (t = n - 1; (a[k] > a[t]) && (t >= k + 1); t--);
  tmp = a[k], a[k] = a[t], a[t] = tmp;
  /* оборачиваем участок массива a[k+1],..., a[n-1] */
  for (i = k + 1; i <= (n + k)/2; i++) {
    t = n + k - i;
    tmp = a[i], a[i] = a[t], a[t] = tmp;
  }
  return i;
}
```

Алгоритм Нарайаны

```
int main (void) {
  int *a, n, i;
  scanf ("%d", &n);
  a = malloc (n * sizeof(int));

  for (i = 0; i < n; i++)
    a[i] = i + 1;
  do {
    PrintPerm (a, n);
  } while (NextPerm (a, n));
  return 0;
}
```

Резюме курса. Введение в теорию алгоритмов.

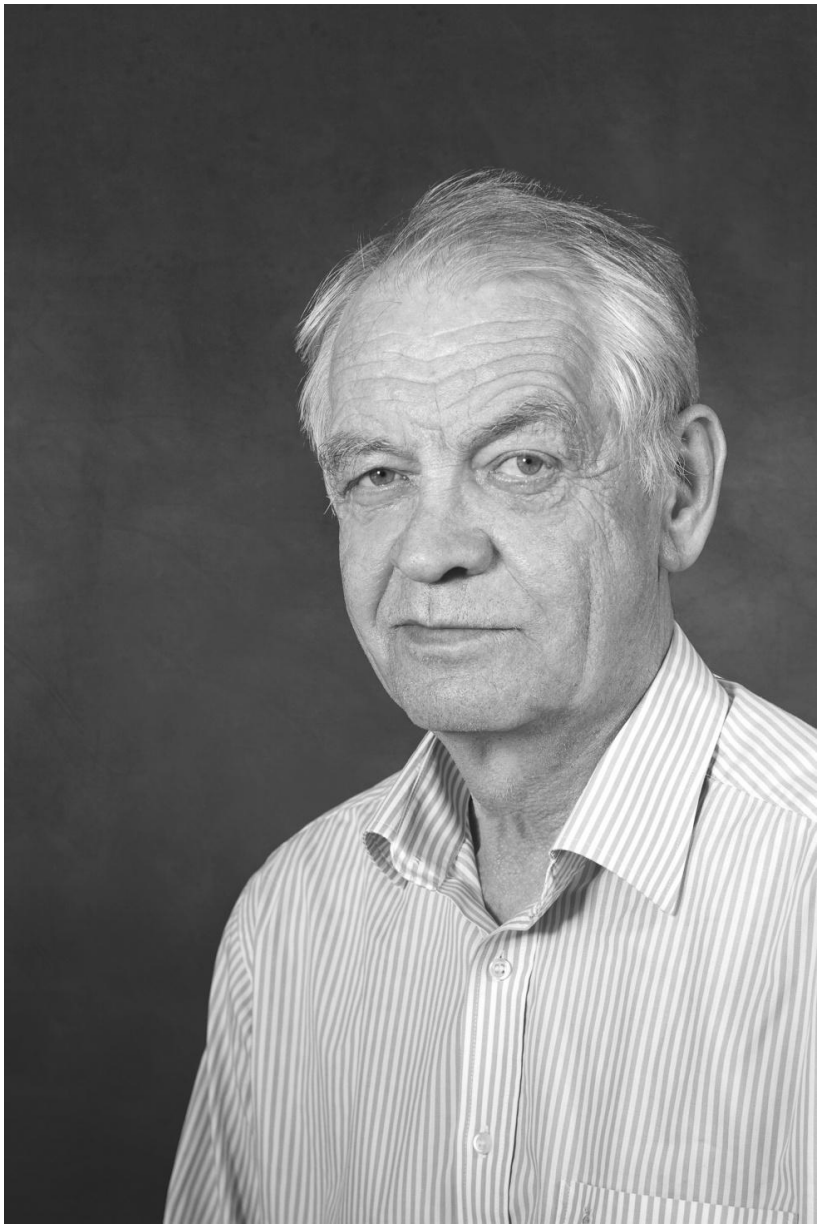
- Формализация алгоритмов: информация (кодирование), исполнители. Связь с задачей обработки информации (частично вычислимыми функциями).
- Возможность построения универсального вычислителя.
- Алгоритмическая неразрешимость.
- Эквивалентность формальных систем описания алгоритмов.

Резюме курса. Язык программирования Си.

- Си-машина. Устройство памяти.
- Особенности, требующие понимания
  - Приведение типов, в том числе integer promotion
  - Точки следования и побочные эффекты
  - «Ленивая» логика
  - Битовые операции
  - Оператор выбора
  - Индексация массивов
  - Строки
  - Адресная арифметика
  - Выравнивание структур
  - Рекурсия (в том числе хвостовая), inline
  - Вызовы по указателю
  - VLA-массивы
  - Динамическая память
  - Программы из нескольких файлов, заголовочные файлы, внешние переменные, компоновка…

Резюме курса. Алгоритмы и структуры данных.

- Списки (варианты «возвратить новый указатель» или «передать двойной указатель»)
- Стеки, очереди
- Сортировка (простые алгоритмы, быстрая сортировка, их сложность). Минимально возможная сложность сортировки.
- Двоичные деревья и их обходы. Замена рекурсии итерацией.
  Прошитые двоичные деревья («посчитать и сохранить» или «пересчитать каждый раз, не хранить»)
- Двоичные деревья поиска. Основные операции и высота дерева.
- Сбалансированные двоичные деревья поиска. Повороты как средство восстановления балансировки. Splay-деревья и чем они отличаются от классических сбалансированных.
  Возможное обобщение сбалансированных деревьев (ранги).
- Хеширование: как разрешать коллизии и как делать хорошие хеш-функции.
- Пирамида и пирамидальная сортировка.
- Дерево цифрового поиска.
- Переборные алгоритмы.



Виктор Петрович Иванников
(1940-2016)