МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ

Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования

«ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ
УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ» (ТУСУР)

Методические указания для выполнения
лабораторных работ
по дисциплине

Программирование

для студентов специальности
080500.62 – «Бизнес-информатика»

Томск – 2012

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ

Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования

«ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ
УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ» (ТУСУР)

Кафедра автоматизации обработки информации

Утверждаю:
Зав. каф. АОИ
профессор
_____Ю.П. Ехлаков
«____» _____2012 г.

Методические указания для выполнения
лабораторных работ
по дисциплине

Программирование

для студентов специальности
080500.62 – «Бизнес-информатика»

Разработчик:
ст.преподаватель каф. АОИ
_____ Н.В. Пермякова
«____» _____2012 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Лабораторный практикум предназначен для закрепления теоретической части дисциплины, основной целью которой является обучение разработке и реализации алгоритмов на основе структурного подхода. Процесс выполнения лабораторных работ направлен на формирование следующих компетенций:

1. Общекультурные компетенции - готовность к ответственному и целеустремленному решению поставленных задач во взаимодействии с обществом, коллективом, партнерами (ОК-7), осознание сущности и значения информации в развитии современного общества; владение основными методами, способами и средствами получения, хранения, переработки информации (ОК-12), способность работать с информацией из различных источников (ОК-16). Для формирования данных компетенций применяются интерактивные формы проведения занятий «Работа в команде (эвристическая беседа)», «Исследовательский метод», «Тестирование» на лекционных занятиях. Процесс формирования общекультурных компетенций контролируется по результатам проверки умения решать поставленные задачи как индивидуально, так и в коллективе, а так же, представлять полученные результаты в виде отчетов по лабораторным работам. В течение семестра регулярно проводятся контрольные работы и тестовые опросы.

2. Профессиональные компетенции - управление контентом предприятия и Интернет-ресурсов, управление процессами создания и использования информационных сервисов (контент-сервисов) (ПК-7), разработка контента и ИТ-сервисов предприятия и Интернет-ресурсов (ПК-18), использование соответствующего математического аппарата и инструментальных средств для обработки, анализа и систематизации информации по теме исследования (ПК-20). Формирование профессиональных компетенций предполагает использование интерактивных форм «Работа в команде (эвристическая беседа)» и «Исследовательский метод» во время проведения лабораторных работ и выполнения индивидуального задания. Формирование профессиональных компетенций контролируется во время выполнения и защиты лабораторных работ, где оцениваются основные качества программных приложений – функциональные возможности, надежность, эффективность и др.

В результате выполнения лабораторных работ студент должен научиться реализовывать описанные различными способами классические алгоритмы на языке Си, студенты должны ознакомиться с основными типами и структурами

данных, используемыми в программировании и использовать их при выполнении лабораторных работ.

Представление множеств упорядоченными списками

Цель работы

Программно реализовать представление множеств упорядоченными списками и основные операции над множествами.

Представление множеств упорядоченными списками

Если рассматриваемое множество не велико, то с точки зрения экономии памяти, множества достаточно часто представляются в виде списков элементов. Элемент списка в этом случае представляется записью с двумя полями: информационным и указателем на следующий элемент. Весь список описывается указателем на первый элемент.

Эффективная реализация операций над множествами, представленными в виде упорядоченных списков, основана на общем алгоритме типа слияния. Общая идея алгоритма типа слияния состоит в следующем: алгоритм параллельно просматривает два множества, представленных упорядоченными списками, причем на каждом шаге продвижение происходит в том множестве, в котором текущий элемент меньше.

Поверка включения слиянием

Даны проверяемые множества A и B, которые заданы указателями a и b.
Если $A \subset B$, то функция возвращает 1, в противном случае 0.

1. *Pa = a; Pb = b;*
2. Пока (*Pa != NULL and Pb != NULL*)
   2.1. Если (*Pa.i<Pb.i*) // сравнение текущих информационных
   // полей.
   То вернуть 0 // элемент множества A отсутствует
   //в множестве B
   Иначе Если (*Pa.i >Pb.i*)
   То *Pb = Pb.n* //перейти к следующему
   // элементу в множестве B.
   Иначе *Pa:=Pa.n; Pb:=Pb.n* // Выполнить переход
   // в обоих множествах
3. Конец цикла
4. Если (*Pa = NULL)* То вернуть 1 // на выходе 1, если достигнут
   Иначе вернуть 0 // конец списка A.
5. Конец

Вычисление объединения слиянием

Даны объединяемые множества A и B, которые заданы указателями a и b.

1. *Pa = a; Pb = b; c = NULL;*

2. Пока (*Pa != NULL and Pb!=NULL*)
2.1. Если (*Pa.i<Pb.i*) // сравнение информационных полей
// текущих элементов
То *d = Pa.i; Pa = Pa.n* // добавлению подлежит элемент
// множества А
Иначе Если (*Pa.i >Pb.i*)
То *d = Pb.i; Pb = Pb.n* // добавлению подлежит
// элемент множества В
Иначе *d = Pb.i ; Pa = Pa.n; Pb = Pb.n*
// в этом случае *Pa.i = Pb.i*, можно взять любой
// элемент
2.2. Добавить *d* в конец списка *c*.
3. Конец цикла
4. Если (*Pa != NULL*)
То добавить в конец списка *c* оставшиеся элементы из *A*
5. Если (*Pb != NULL*)
То добавить в конец списка *c* оставшиеся элементы из *B*
6. Конец

Вычисление пересечения слиянием

Даны пересекаемые множества *A* и *B*, которые заданы указателями *a* и *b*.

1. *Pa = a; Pb = b; c = NULL;*
2. Пока (*Pa != NULL* and *Pb!=NULL*)
2.1. Если (*Pa.i < Pb.i*) // сравнение текущих информационных
// полей.
То *Pa = Pa.n* // элемент множества А не принадлежит
// пересечению
Иначе Если (*Pa.i >Pb.i*)
То *Pb = Pb.n* // элемент множества В не
// принадлежит пересечению
Иначе *d = Pb.i; Pa = Pa.n; Pb =Pb.n* // в этом случае
// *Pa.i = Pb.i,* можно взять любой элемент
2.2. Добавить *d* в конец списка *c*.
3. Конец цикла
4. Конец

Задание на выполнение

Даны два множества *A* и *B*. Организовать представление множеств в виде линейных однонаправленных списков. Мощность множеств и элементы множеств задавать с клавиатуры. В программе выполнить проверку списка на упорядоченность и на уникальность элементов.

1. Проверить, включено ли множество *A* во множество *B*.
2. Найти пересечение множеств *A* и *B*.
3. Найти объединение множеств *A* и *B*.

Машинное представление графов

Цель работы

Разработка и реализация алгоритмов преобразования различных форм представления графов.

Машинные способы представления графов

Матрица смежности

Матрицей смежности неориентированного графа *G=(X,U)*, $|X| = n, |U| = m$ называется квадратная матрица $A[n \times n]$, элементы которой задаются по правилу 1.

$$a_{i,j} = \begin{cases} 1, & \text{если вершина } x_i \text{ смежна вершине } x_j; \\ 0, & \text{если вершина } x_i \text{ не смежна вершине } x_j; \end{cases} \qquad (1)$$

Матрицей смежности ориентированного графа *G=(X,U)*, $|X| = n, |U| = m$ называется квадратная матрица $A[n \times n]$, элементы которой задаются по правилу 2.

$$a_{i,j} = \begin{cases} 1, & \text{если вершина } x_i \text{ смежна вершине } x_j; \\ -1, & \text{если вершина } x_j \text{ смежна вершине } x_i; \\ 0, & \text{если вершина } x_i \text{ не смежна вершине } x_j; \end{cases} \qquad (2)$$

Матрица инцидентности

Матрицей инцидентности неориентированного графа *G=(X,U)*, $|X| = n, |U| = m$ называется матрица $B[n \times m]$, элементы которой задаются по правилу 3.

$$b_{i,j} = \begin{cases} 1, & \text{если вершина } x_i \text{ инцидентна ребру } u_j; \\ 0, & \text{если вершина } x_i \text{ не инцидентна ребру } u_j. \end{cases} \qquad (3)$$

Матрицей инцидентности ориентированного графа *G=(X,U)*, $|X| = n, |U| = m$ называется матрица $B[n \times m]$, элементы которой задаются по правилу 4.

$$a_{i,j} = \begin{cases} 1, \text{ если вершина } x_i \text{ начало дуги } u_j; \\ -1, \text{ если вершина } x_i \text{ конец дуги } u_j; \\ 2, \text{ если } u_j \text{ - петля при вершине } x_i; \\ 0, \text{ если вершина } x_i \text{ не инцидентна дуге } u_j. \end{cases} \quad (4)$$

Список ребер

Списком ребер неориентированного графа *G=(X,U)*, $|X| = n, |U| = m$ называются два массива $N[2m]$ и $K[2m]$, элементы которых задаются по правилу 5:

$n_i -$ вершина, являющаяся началом ребра с номером *i.*
$k_i -$ вершина, являющаяся концом ребра с номером *i.* (5)

Списком ребер ориентированного графа *G=(X,U)*, $|X| = n, |U| = m$ называются два массива $N[m]$ и $K[m]$, элементы которых задаются по правилу 6:

$n_i -$ вершина, являющаяся началом дуги с номером *i.*
$k_i -$ вершина, являющаяся концом дуги с номером *i.* (6)

Структура смежности

Структурой смежности графа *G=(X,U)*, $|X| = n, |U| = m$ называется массив списков

$x_i : x_k, x_{k1}, ... x_{kz}$, $i = \overline{1,n}$,

где $x_k, x_{k1}, ... x_{kz}$ - все вершины, смежные вершине x_i.

Пример реализации структуры смежности предложен в следующей программе:

```
#include <iostream.h>
#include <conio.h>

void main()
{
struct List {
```

```
  int Number;
  List *Next;
  };

List *Smegn;  // массив вершин
int n; // количество вершин графа

clrscr();
cout << "Введите количество вершин графа: ";
cin >> n;
// Выделение памяти под массив вершин
Smegn = new List [n];
for (int i=0;i<n;i++)
  {
   Smegn[i].Number = i+1;
   Smegn[i].Next = NULL;
  }
// Ввод структуры смежности
cout << "Признак окончания ввода - 0" << endl;
for(i = 0;i<n;i++)
 {
 cout << "Вводите вершины смежные вершине " << i+1 << " : ";
 int d = 1;
 List* Cur = &Smegn[i];
 while (d!=0)
   {
            cout << "# вершины: ";
            cin >> d;
            if (d==0) continue;
            Cur->Next = new List; // выделение памяти под новый элемент
            Cur = Cur->Next;
            Cur->Next = NULL;
            Cur->Number = d;
   }
 }
// Печать структуры смежности
for (i=0;i<n;i++)
  {
   cout << Smegn[i].Number << ": "; // номер вершины
   List *Cur = &Smegn[i];
   if (Cur->Next == NULL) {cout << endl; continue;}; // Если смежных нет, то
                                                     //перейти на следующую
                                                     //вершину.
   do // пока не пройдены все смежные вершины,
      //выводить на экран номера вершин
            {
             if (Cur->Next->Next == NULL) continue;
```

```
            Cur = Cur -> Next;
            cout << Cur->Number << ",";
           }
           while (Cur->Next->Next != NULL);
            Cur = Cur->Next;
            cout << Cur->Number << "." << endl; // вывод последней вершины
  }
cout << endl;
// удаление структуры смежности из памяти.
for (i=0;i<n;i++)
{
 List *Top;
 Top = &Smegn[i];
 List* Cur = Top->Next;
 delete Top;
 Top = Cur;
 while (Cur!=NULL)
  {
 Cur = Top->Next;
 delete Top;
 Top = Cur;
  }
  delete [] Smegn;
  }
getch();
}
```

Задание на выполнение

Написать программу по заданному варианту. Ввод данных выполнить с клавиатуры или из текстового файла. Предусмотреть ошибки ввода – некорректные данные, отсутствие файла и т.д..

Алгоритмы порождения комбинаторных объектов

Цель работы

Ознакомиться с алгоритмами, генерирующими комбинаторные объекты и программно реализовать их.

Генерация сочетаний

Генерация сочетаний в лексикографическом порядке

Будем рассматривать в качестве множества $X = \{1,2,...,n\}$. Требуется сгенерировать все подмножества мощности $k, (0 \le k \le n)$ множества *X*.

Определим отношение лексикографического порядка ($\prec$) следующим образом. Пусть $a = (a_1, a_2, ..., a_n)$, $b = (b_1, b_2, ..., b_m)$. Будем говорить, что набор *a* предшествует набору *b*: $a \prec b \Leftrightarrow \exists r \ge 1 : a_r < b_r$ и $\forall i = \overline{1, r-1}$; $a_i = b_i$.

Будем рассматривать сочетания *k* элементов из множества *X* как вектор $(c_1, c_2, ..., c_k)$, компоненты которого расположены в порядке возрастания слева направо (т.е. $c_i < c_{i+1}$ для любого *i*). Начиная с сочетания $(1,2,...,k)$, следующие будем строить, просматривая текущее справа налево, чтобы найти самый первый элемент, не достигший максимального значения; этот элемент увеличим на единицу, а всем элементам справа от него присвоим номинальные наименьшие значения.

Лексикографический порядок порождения сочетаний не является алгоритмом с минимальными изменениями.

1. $c_0 := -1$
2. Цикл ($i := \overline{1,k}$)
   2.1. $c_i := i$
3. Конец цикла
4. $j := 1$
5. Пока ($j \ne 0$)
   5.1.Печать $(c_1, c_2, ..., c_k)$
   5.2. $j := k$
   5.3. Пока ($c_j = n - k + j$)
      5.3.1 $j := j - 1$
   5.4. Конец цикла
   5.5. $c_j := c_j + 1$
   5.6.Цикл $(i := \overline{j+1, k})$
      5.6.1. $c_i := c_{i-1} + 1$
   5.7. Конец цикла
6. Конец цикла
7. Конец

Генерация сочетаний с помощью кодов Грея

При генерации сочетаний из *n* элементов по *k* наименьшим возможным изменением при переходе от текущего сочетания к следующему является замена одного элемента другим. В терминах Грея это означает, что мы хотим

выписать все *n*-разрядные кодовые слова, содержащие ровно *k* единиц, причем последовательные наборы отличаются ровно в двух разрядах (в одном из разрядов 0 заменяется на 1, а в другом — 1 на 0).

Пусть $G(n)$ — двоично-отраженный код Грея, а $G(n,k)\ (0 \le k \le n)$ — последовательность кодовых слов ровно с *k* единицами:

$$G(n,k)^T = \left(G(n,k)_1, G(n,k)_2, ..., G(n,k)_{C_n^k}\right)^T.$$

Эту последовательность можно рекурсивно определить следующим образом:

$$G(n,0) = (0\ 0\ ...\ 0);$$

$$G(n,n) = (1\ 1\ ...\ 1);$$

$$G(n,k) = \begin{pmatrix} 0 & G(n-1,k) \\ 1 & \overline{G(n-1,k-1)} \end{pmatrix}, \qquad (7).$$

где 0 — вектор-столбец размерности $C_{n-1}^k \times 1$, состоящий из нулей;

1 — вектор-столбец размерности $C_{n-1}^{k-1} \times 1$, состоящий из единиц;

$G(n-1,k)$ — матрица $C_{n-1}^k \times (n-1)$ кодовых слов, содержащих ровно *k* единиц;

$\overline{G(n-1,k-1)}$ — матрица $C_{n-1}^{k-1} \times (n-1)$ кодовых слов, содержащих ровно $k-1$ единиц, причем кодовые слова записаны в порядке, обратном порядку $G(n-1,k-1)$ ($\overline{G}$ — «перевернутая» матрица *G*).

На рис 1. приведен пример построения кодовых слов Грея для генерации сочетаний из 4 элементов по 2.

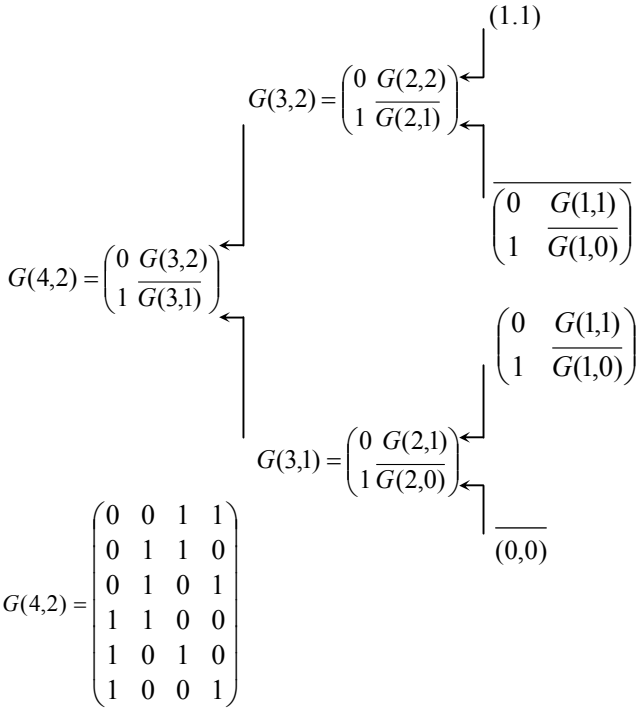


Рис. 1. Кодовые слова Грея для сочетаний из 4 по 2

Индукцией по n доказывается, что последовательность кодовых слов $G(n,k)$ получается удалением из кода Грея $G(n)$ всех кодовых слов с числом единиц, не равным k, причем в этой последовательности любые два соседних кодовых слов различаются только в двух позициях.

Генерация перестановок

Генерация перестановок в лексикографическом порядке

Будем рассматривать исходное множество $X=\{1,2,...,n\}$, и в качестве начальной перестановки возьмем $\pi'=(1,2,...,n)$. Условие окончания работы — порождение перестановки $\pi''=(n,n-1,...,2,1)$, которая является последней в лексикографическом смысле среди всех перестановок множества X. Переход

от текущей перестановки $\pi = (\pi_1, \pi_2, ..., \pi_n)$ к следующей за ней будем осуществлять таким образом:

1) просматривая перестановку π справа налево, ищем самую первую позицию *i* такую, что $\pi_i < \pi_{i+1}$ (если такой позиции нет, значит текущая подстановка $\pi = \pi''$ и процесс генерации завершается);

2) просматривая π от π_i слева направо, ищем наименьший из элементов π_j такой, что $\pi_1 < \pi_j (i < j)$;

3) меняем местами элементы π_i и π_j; затем все элементы $\pi_{i+1}, \pi_{i+2}, ..., \pi_n$ записываем в обратном порядке (т.е. меняем местами симметрично расположенные элементы π_{i+1+t} и π_{n-t}).

Пример. Пусть текущая перестановка π имеет вид $\pi = (3,5,7,6,4,2,1)$. На первом шаге найдены $\pi_i = 5, i = 2$; на втором — $\pi_j = 6, j = 4$; на третьем шаге меняем местами π_i и π_j: (3, 6, 7, 5, 4, 2, 1) и меняем местами элементы, начиная с третьей позиции: (3, 6, 1, 2, 4, 5, 7) — получили подстановку, следующую за текущей в лексикографическом порядке.

Генерация перестановок с помощью вложенных циклов

Будем говорить, что перестановка $\pi = \begin{pmatrix} 1 & 2 & ... & n \\ a_1 & a_2 & ... & a_n \end{pmatrix}$ является циклом длины *k* степени *d*, если ее элементы $a_i, i = \overline{1,k}$, получены из 1, 2, … , *k* циклическим сдвигом вправо на *d* позиций, остальные *n* – *k* элементов стационарны. Например, подстановка $\pi = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 4 & 1 & 2 & 3 & 5 & 6 \end{pmatrix}$ является циклом длины 4 степени 1.

Алгоритм порождения подстановок с помощью вложенных циклов основан на следующей теореме.

Теорема 1. ***Любую подстановку*** π ***на множестве*** $X = \{1,2,...,n\}$ ***можно представить в виде композиции***

$$\pi = \rho_n \circ \rho_{n-1} \circ ... \circ \rho_1 \qquad (8)$$

где ρ_i ***— циклическая подстановка порядка*** *i*.

Пример. Представим в виде (8) подстановку $\pi = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 3 & 2 & 4 & 1 \end{pmatrix}$, т.е. запишем

$$\pi = \rho_4 \circ \rho_3 \circ \rho_2 \circ \rho_1 = \begin{pmatrix} 1 & 2 & 3 & 4 \\ a_1 & a_2 & a_3 & a_4 \end{pmatrix} \circ \begin{pmatrix} 1 & 2 & 3 & 4 \\ b_1 & b_2 & b_3 & 4 \end{pmatrix} \circ$$

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ c_1 & c_2 & 3 & 4 \end{pmatrix} \circ \begin{pmatrix} 1 & 2 & 3 & 4 \\ d_1 & 2 & 3 & 4 \end{pmatrix}.$$

Очевидно, последний цикл является тождественной подстановкой. Определим ρ_4 : т.к. $\rho_3(4) = 4, \rho_2(4) = 4,$ то $\rho_4(4) = 1$ следовательно

$$\rho_4 = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 1 & 4 \end{pmatrix} \text{ — цикл порядка 4.}$$

Т.к. $\rho_2(3) = 3$, то $3 = \pi(1) = \rho_3(2) = \rho_3(\rho_4(1)) = \rho_2(2)$ и

$$\rho_2 = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 1 & 3 & 4 \end{pmatrix}.$$

Разложение подстановки π имеет вид:

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 3 & 2 & 4 & 1 \end{pmatrix} \circ \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 1 \end{pmatrix} \circ \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 1 & 4 \end{pmatrix} \circ \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 1 & 3 & 4 \end{pmatrix} \qquad (1.3)$$

Диаграмма композиции (1.3) приведена на рис. 2.

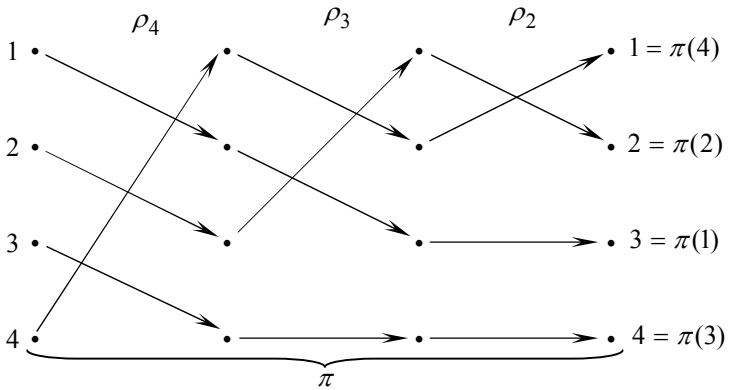


Рис. 2. Разложение в произведение вложенных циклов

Из теоремы 1 следует, что все перестановки можно получить систематическим перебором циклических сдвигов. В качестве начальной перестановки берем $\pi' = (1,2,...,n)$ и сдвигаем на одну позицию вправо все элементы до тех пор, пока вновь не получим π'; теперь сдвигаем циклически первые $n-1$ элементов и снова повторяем сдвиг всех *n* элементов на одну позицию до тех пор, пока не получим уже имеющуюся перестановку; сдвигаем циклически ее

первые $n-2$ элементов… и т.д., пока не переберем все $n!$ перестановок. Ниже приведен алгоритм вложенных циклов.

1. Цикл ($i = \overline{1,n}$)
    1.1. $\pi_i := i$;
2. Конец цикла
3. $k := 0$
4. Пока ($k \neq 1$)
    4.1.Печать $\pi = (\pi_1, \pi_2, ..., \pi_n)$
    4.2. $k := n$
    4.3. Сдвиг первых *k* элементов на одну позицию
    4.4. Пока ($\pi_k = k$ и $k > 0$)
        4.4.1. $k := k - 1$
        4.4.2. Сдвиг первых *k* элементов на одну позицию
    4.5. Конец цикла
5. Конец цикла
6. Конец

Этот алгоритм не является эффективным, т.к. на каждом шаге требует большого количества (не меньше *n*) транспозиций (транспозиция — обмен местами двух элементов).

Транспозиция соседних элементов

Описанные выше алгоритмы генерации перестановок не являются алгоритмами с минимальными изменениями. Минимальным изменением при переходе от текущей перестановки к следующей является транспозиция двух элементов. Дадим рекурсивное описание такого алгоритма.

Если $n = 1$, то существует единственная перестановка $\pi^{(1)} = (1)$. Пусть $n > 1$ и последовательность перестановок $\pi^{(1)}, \pi^{(2)}, ..., \pi^{(r)}, r = (n-1)!$ на множестве $(1,2,...,n-1)$ построена. Для получения перестановок на множестве $(1,2,...,n)$ будем вставлять элемент *n* на «промежутке» между элементами перестановки $\pi^{(i)}$ по следующему правилу: если номер *i* подстановки $\pi^{(i)}$ — нечетное число, то элемент *n* вставляется в промежутки справа налево, если *I* — четное число, то элемент *n* вставляется в промежутки между элементами $\pi^{(i)}$ слева направо.

Пример генерации перестановки при $n = 4$ приведен на рис.3.

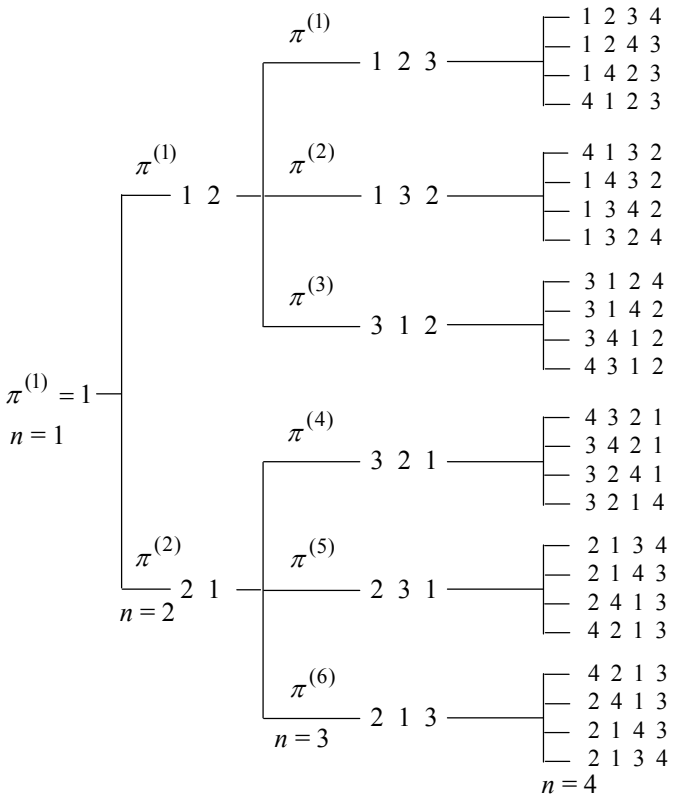


Рис. 3. Генерация перестановок транспозицией соседних элементов ($n = 4$)

Задание на выполнение

Задать с клавиатуры мощность множества n, элементы множества и мощность выборки k. Реализовать алгоритм генерации сочетания

1. в лексикографическом порядке;
2. с помощью кодов Грея.

Задать с клавиатуры мощность множества n и элементы множества. Реализовать алгоритм генерации перестановок:

3. в лексикографическом порядке;
4. с помощью вложенных циклов;

транспозицией соседних элементов.

Алгоритмы на графах

Цель работы

Программная реализация основных алгоритмов на графах

Алгоритмы обходов графа

Дан граф $G=(X,U)$, $|X| = n, |U| = m$.

1. Цикл ($i = \overline{1,n}$)
    1.1. $M[x_i]=0$; // Все вершины не отмечены
2. Конец цикла
3. Выбрать произвольную вершину $v = x_i \in X$
4. $v \to T$ // Записать выбранную вершину в структуру T
5. $M[v] = 1$ // Отмечаем вершину, как пройденную
6. Пока (T не пуста)
    6.1. $u \leftarrow T$ // извлекаем вершину из структуры
    6.2. Печать u
    6.3. Цикл(для всех вершин w, смежных с u)
        6.3.1. Если ($M[w]=0$) То $w \to T$; $M[w]=1$
    6.4. Конец цикла
7. Конец цикла
8. Конец

Если структуру T определить как стек, то алгоритм обхода будет выполняться «в глубину». Если же T определена как очередь, будет выполняться обход «в ширину».

Алгоритмы поиска путей на графах

Алгоритм Дейкстры

Алгоритм Дейкстры находит кратчайший путь между двумя заданными вершинами в орграфе.

Дан граф $G=(X,U)$, $|X|=n, |U|=m$. Граф задан матрицей весов C, s– начальная вершина обхода, t – конечная вершина обхода.

На выходе алгоритма создаются два массива:

- T – если вершина v лежит на кратчайшем пути, то $T[v]$- длина кратчайшего пути от s к v.
- H – $H[v]$ – вершина, непосредственно предшествующая v на кратчайшем пути.

1. Цикл($x=\overline{1,n}$)
   1.1. $T[x]=\infty$ // кратчайший путь не известен
   1.2. $M[x]=0$ // все вершины не отмечены
2. Конец цикла
3. $H[s]=0$ // s ничего не предшествует
4. $T[s]=0$ // кратчайший путь имеет длину 0
5. $M[s]=1$ // вершина s пройдена
6. $v=s$ // зафиксируем текущую вершину
7. Цикл (1)
   7.1. Цикл(для $\forall u$ смежных с v)
      7.1.1. Если ($M[u]==0$ и $T[u]>T[v]+C[v,u]$)
         То $T[u]>T[v]+C[v,u]$; // найден более
         // короткий путь из s в u через v
         $H[u]=v$;
   7.2. Конец цикла
   7.3. $t=\infty$
   7.4. $v=0$
   7.5. Цикл($x=\overline{1,n}$)
      7.5.1. Если ($M[x]==0$ и $T[x]<t$)
         То $v=u$; $t=T[u]$ // вершина v заканчивает
         // кратчайший путь из s.
   7.6. Конец цикла
   7.7. Если ($v==0$) То «Нет пути из s в t»; Закончить выполнение цикла.
   7.8. Если ($v==t$) То «Путь найден»;
      Печать $T[t]$;
      Печать $H[t]$.
   7.9. $M[v]=1$
8. Конец цикла
9. Конец

Алгоритм Форда

Алгоритм Форда позволяет найти расстояние от заданной вершины s до всех вершин $T[v]=d(s,v)$, ориентированного графа. Исходными данными для этого алгоритма является матрица весов дуг C. В отличие от алгоритма Дейкстры алгоритм может быть использован на графах с отрицательными длинами дуг.

Дан граф $G=(X,U)$, $|X|=n, |U|=m$. Граф задан матрицей весов C. s – вершина начала пути.

1. Цикл (для всех вершин x графа G)
   1.1. $D[x]:=C[s,x];$
2. Конец цикла
3. $D[s]:=0;$
4. Цикл ($k=\overline{1,n-2}$)

// последовательно перебрать все ребра

   4.1. Цикл (для всех вершин v графа, $v \neq s$)

   // применить процесс поиска кратчайшего

   //пути для всех ребер

      4.1.1. Цикл (для всех вершин u графа, $u \neq v$)

         4.1.1.1 Если ($D[v]>D[u]+C[u,v]$)

            То $D[v]=D[u]+C[u,v]$

      4.1.2. Конец цикла

   4.2. Конец цикла

// закончить алгоритм досрочно

5. Если нет изменений в D, То $k=n-2$
6. Конец цикла
7. Печать D.
8. Конец

Алгоритм ближайшего соседа

Существует другой метод получения кратчайшего остовного дерева, который не требует ни сортировки ребер, ни проверки на цикличность на каждом шаге, - так называемый *алгоритм ближайшего соседа*. Просмотр начинается с некоторой произвольной вершины a в заданном графе. Пусть (a,b)- ребро с наименьшим весом, инцидентное a; ребро (a,b) включается в остов. Затем среди всех ребер, инцидентных либо a, либо b, выбираем ребро с наименьшим весом и включаем его в частично построенное дерево. В результате этого в дерево добавляется новая вершина, например, c. Повторяя процесс, ищем наименьшее ребро, соединяющее a, b или c с некоторой другой вершиной графа. Процесс продолжается до тех пор, пока все вершины из G не будут включены в дерево, то есть пока дерево не станет остовным.

Дан граф $G=(X,U)$, $|X|=n, |U|=m$. Граф задан матрицей весов C.

1. T – пустое множество ребер остовного дерева
2. a – произвольная вершина графа
3. $a \to T$
4. Пока ($T \neq X$)
   4.1. Найти ребро (u,v) с минимальным весом, такое, что $u \in T, v \notin T$.
   4.2. $v \to T$

5. Конец цикла
6. Печать *T*
7. Конец

Алгоритм Краскала

Алгоритм Краскала относится к семейству «жадных» алгоритмов. Алгоритм ищет кратчайший остов в связном графе *G=(X,U),* $|X| = n, |U| = m$. Граф задан списком ребер U с длинами. На выходе алгоритма создается список T ребер кратчайшего остова.

1. *T* – пустой список
2. Упорядочить список *U* в порядке возрастания длин
3. *k = 1 //* номер рассматриваемого ребра
4. Цикл($i = \overline{1, m-1}$)
   4.1. Пока (добавление ребра *E[k]* образует цикл в *T*)
      4.1.1. *k=k+1*
   4.2. Конец цикла
   4.3. $E[k] \to T$
5. Конец цикла
6. Печать *T*
7. Конец

Волновой алгоритм

Волновой алгоритм является алгоритмом поиска кратчайшего пути между двумя вершинами в неориентированном ненагруженном графе.

Пусть дан граф *G=(X,U),* $|X| = n, |U| = m$. Вершина *a* – начало пути, вершина *b* – конец пути. Запишем вершину *a* во фронт волны нулевого уровня FW_0. Введем переменную *i =0.* Далее найдем все вершины, смежные вершинам $v \in FW_i$ и ранее не пройденные. Запишем эти вершины во фронт волны следующего уровня FW_{i+1}. Увеличим *i* - *i = i +1.* Если среди найденных вершин есть вершина *b,* то длина кратчайшего пути найдена и равна *i*, если нет, то процесс продолжается до тех пор пока не будет найдена конечная вершина пути, либо, пока не будут просмотрены все вершины графа. В этом случае пути из *a* в *b* нет.

1. Цикл($i = \overline{1, n}$)
   1.1. $M[x_i] = -1$ // отметим все вершины, как не пройденные
2. Конец цикла
3. *M[a]=0; //* вершина пройдена
4. *i=1*
5. $a \cup FW_{i-1}$

6. Пока ($b \notin FW_{i-1}$ И $\exists x \in X, M[x] = -1$)

6.1. Цикл (для вершин $v \in X, M[v] = -1$ и смежных вершинам $\notin FW_{i-1}$)

6.1.1. $v \cup FW_i$

6.1.2. $i = i + 1$

6.2. Конец цикла

7. Конец цикла

8. Если $b \in FW_{i-1}$ То «Путь найден», длина пути равна *i-1*.
Иначе «Путь не существует»

9. Конец

Алгоритм Уоршалла

Алгоритм вычисления транзитивного замыкания для орграфа *G=(X,U),* $|X| = n, |U| = m$. Граф задан матрицей смежности.

1. $S = R$ // вспомогательная матрица

2. Цикл($i = \overline{1,n}$)

2.1. Цикл($j = \overline{1,n}$)

2.1.1. Цикл($k = \overline{1,n}$)

$T[i, j] = S[j, k] \vee S[j, i] \& S[i, k]$ // вычисление //матрицы // транзитивного // замыкания

2.2. Конец цикла

3. Конец цикла

4. Печать *T*.

5. Конец

Алгоритм построения эйлеровой цепи

Если граф имеет цикл содержащий все ребра графа по одному разу, то такой цикл называется эйлеровым циклом, а граф называется эйлеровым графом. Если граф имеет цепь (не обязательно простую), содержащую все вершины по одному разу, то такая цепь называется эйлеровой цепью, а граф называется полуэйлеровым графом.

Для того, чтобы в графе существовал эйлеров цикл граф должен быть связанным, для неориентированных графов число ребер в каждой вершине должно быть четным.

Дан эйлеров граф $G=(X,U)$, $|X|=n, |U|=m$, заданный структурой смежности. $\Gamma[v]$- множество вершин, смежных с вершиной v.

1. S – пустой стек //стек для хранения вершин
2. Выбрать произвольную вершину $x \in X$
3. $x \to S$ // положить x в стек
4. Пока (Стек не пуст)
   4.1. $v \leftarrow S$
   4.2. $v \to S$
   4.3. Если $\Gamma[v]$- пустое множество То $v \leftarrow S$; Печать v
   Иначе взять $u \in \Gamma[v]$; // взять первую
   // вершину, смежную v
   $u \to S$
   //удалить ребро (v,u)
   $\Gamma[v]:=\Gamma[v]\backslash u$;
   $\Gamma[u]:=\Gamma[u]\backslash v$;
5. Конец цикла
6. Конец

Задание на выполнение

Программно реализовать один из описанных алгоритмов. Программа должна позволять пользователю вводить данные о графе с клавиатуры, либо считывать из текстового файла.

Простые сортировки на месте

Цель работы

Ознакомиться и реализовать простые методы сортировки – сортировки обменом, выбором, вставками, бинарными вставками.

Сортировка обменом

Одним из простых методов сортировки на месте (т.е. при работе не требуется дополнительный объем памяти) является сортировка обменом или «пузырьковая» сортировка.

Суть алгоритма состоит в следующем: сравниваются пары рядом стоящих элементов массива, если первый элемент пары меньше второго, то элементы меняются местами. После первого просмотра массива самый большой элемент встает на свое место, а маленькие по значению элементы на один шаг продвигаются к началу массива. Отсюда метод и получил свое название: «легкие» элементы плавно «всплывают» к началу массива. «Тяжелые» элементы быстро «тонут», встают в конец массива.

После того, как все пары элементов массива просмотрены, массив еще не будет отсортирован, поэтому необходимо просмотреть пары элементов еще раз, и тогда еще один самый большой элемент встанет на свое место.

Если массив состоит из *n* элементов, то пар в таком массиве ровно *n-1*. На каждом шаге алгоритма самый большой элемент массива становится на свое место. Поэтому количество просматриваемых пар уменьшается с каждым шагом на единицу.

Таким образом, в алгоритме явно просматриваются два вложенных цикла. Внутренний цикл отвечает за просмотр пары элементов, количество шагов этого цикла зависит от внешнего цикла. Чем больше шагов выполнил внешний цикл, тем меньше пар просматривает внутренний цикл. Т.к. при выполнении одного шага внешнего цикла самый большой элемент становится на место, то количество шагов цикла равно количеству элементов массива *-1*. Однако, массив может быть уже отсортированным, до окончания работы внешнего цикла. Можно досрочно остановить выполнение цикла, если на каком-то *i*- том шаге во внутреннем цикле не произошло ни одного обмена.

Покажем применение сортировки на массиве *1 5 2 4 3.*

1: 1 2 4 3 ***5***

2: 1 2 3 ***4 5***

3: ***1 2 3 4 5***

После третьего шага алгоритм может закончить свою работу, так как не было проведено ни одного обмена.

Сортировка выбором

Сортировка выбором использует другой принцип упорядочивания элементов. На начальном шаге алгоритма выбирается минимальный элемент и ставится на место нулевого элемента. На последующих, *i*- тых шагах выбирается минимальный элемент среди элементов от *i*- того до (*n-1)*-го.

Рассмотрим на примере массива *1 5 2 4 3.*

1: ***1*** *5 2 4 3* Первый минимальный элемент уже стоит на своем месте.

2: ***1 2*** *5 4 3* Второй минимальный элемент – 2, поменяем его местами с 5.

3: ***1 2 3*** *4 5* Третий минимальный элемент – 3, поменяем его местами с 5.

4: ***1 2 3 4*** *5* Четвертый минимальный элемент стоит на своем месте.

После этого сортировка заканчивается, т.к. последний элемент в любом случае будет стоять на своем месте.

Сортировка вставками

Сортировка вставками упорядочивает массив, выполняя следующие действия – на начальном шаге алгоритма считаем, что последовательность из одного, первого элемента упорядоченная последовательность. Найдем место в этой последовательности для второго элемента. После этого упорядочены уже первые два элемента. Далее в текущей упорядоченной последовательности ищутся места для третьего, четвертого и т.д. элементов.

При программировании поиск места для вставляемого элемента лучше всего осуществить с конца упорядоченной последовательности.

Рассмотрим на примере массива *1 5 2 4 3.*

1: ***1 5*** *2 4 3* В упорядоченную последовательность добавляется 5.
2: ***1 2 5*** *4 3* В упорядоченную последовательность добавляется 2.
3: ***1 2 4 5*** *3* В упорядоченную последовательность добавляется 4.
4: ***1 2 3 4 5*** В упорядоченную последовательность добавляется 3.

Сортировка бинарными вставками

Сортировка бинарными вставками некоторым образом улучшает предыдущий алгоритм, увеличивая скорость нахождения места для вставляемого элемента. Метод использует информацию о том, что часть выборки уже отсортирована. Тогда, можно организовать поиск места для вставки нового элемента следующим образом: пусть переменная F обозначает индекс начала упорядоченной последовательности, а L - индекс конца. Найдем индекс элемента, находящегося в центре отсортированной последовательности - $M = (F + L)/2$. Сравним вставляемый элемент *X[j]* с элементом *X[M]*. Если *X[j]> X[M]*, то изменим F на M (будем искать в правой части последовательности). Если *X[j]< =X[M]*, то изменим L на M (будем искать в левой части последовательности). Описанные действия будем проводить до тех пор, пока *L>F*. После выхода из цикла место для вставки элемента найдено.

Рассмотрим работу алгоритма на примере уже упорядоченной последовательности:

1 2 3 5 6 7 8 9 10 ***4.***

F=0, L = 8, M = 4 X[4]=6 > 4, следовательно:

F = 0 L = 4 M = 2 X[2]=3 < 4, следовательно:

F = 2 L = 4 M = 3 X[3]=5 >4, следовательно:

F = 2, L = 3 M = 2 (по правилам деления целых чисел в Си 5/2 = 2) X[2]= 3 < 4, следовательно F = 3 L = 3.

Так как F и L равны (условие выполнения цикла), то вставляемый элемент должен занимать в упорядоченной последовательности место с номером 3.

Задание на выполнение

- Получите вариант индивидуального задания у преподавателя.
- Составьте алгоритм простой сортировки.
- Реализуйте алгоритм на языке Си, добавив в программу подсчет количества сравнений и перестановок, проведенных алгоритмом.

Выполните полученную программу на случайных массивах размерности 100, 1000, 10000.

Улучшенные сортировки

Цель работы

Ознакомиться и реализовать один из улучшенных методов сортировок – пирамидальной, Шелла, комбинированной, Хоара. Сравнить время работы простой сортировки и улучшенной.

Пирамидальная сортировка

Пирамида – это частично упорядоченное двоичное дерево, элементы которого расположены в узлах дерева по следующему правилу – каждый элемент родительского узла больше или равен элементов, расположенных в дочерних узлах. Следующий рисунок представляет пирамиду из 15 элементов:

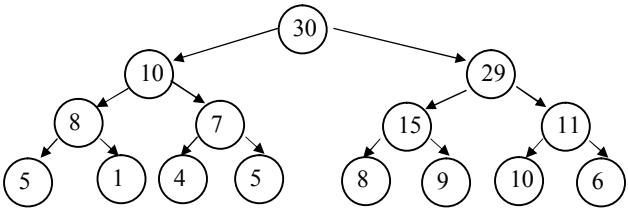


Рис. 4. Пирамида из 15 элементов.

Элементы дерева легко представляются в виде массива – пусть родительский узел имеет индекс i, тогда дочерние узлы имеют индексы $2i+1$ и $2i+2$.

Рассмотренная пирамида может быть представлена массивом:

Индексы	0	1	2	3	4	5	6	7
Элементы	30	10	29	8	7	15	11	5
Индексы	8	9	10	11	12	13	14	
Элементы	1	4	5	8	9	10	6	

Т.к. корневой элемент элемент пирамиды всегда является максимальным элементом, то процесс пирамидальной сортировки можно описать следующим

образом: поменять верхний элемент пирамиды с нижним элементом и рассматривать в дальнейшем не n элементов исходного массива, а $n-1$ элемент. Но при обмене элементов нарушается правило расположения элементов в пирамиде, поэтому после обмена необходимо перестроить пирамиду с $n-1$ элементами и повторять два этих шага, пока пирамида не останется пустой. Таким образом, необходимо написать процедуру, строящую пирамиду для произвольного массива размерности n, далее алгоритм пирамидальной сортировки очень прост.

Рассмотрим процесс построения пирамиды на произвольном массиве:

Индексы	0	1	2	3	4	5	6	7
Элементы	25	11	5	11	4	8	3	28
Индексы	8	9	10	11	12	13	14	
Элементы	18	10	1	5	4	2	17	

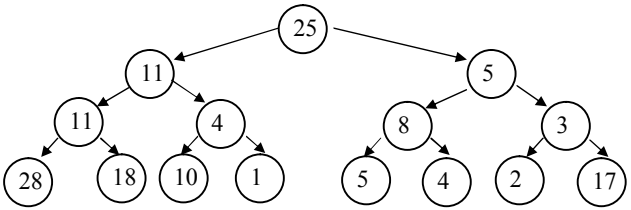


Рис. 5. Произвольный массив из 15 элементов.

N = 15. Для элементов, находящихся на нижнем уровне не существует дочерних элементов, т.е. эти элементы могут не проверяться на выполнение правила пирамиды, индексы этих элементов лежат в интервале от *N/2* до *N*. Поэтому построение начинается с элемента с номером *N/2-1*, в примере это *x[6] = 3*, сравним этот элемент с наибольшим из элементов *x[13]* и *x[14]:*

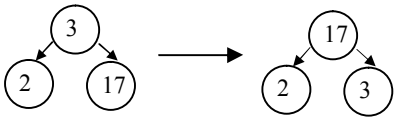
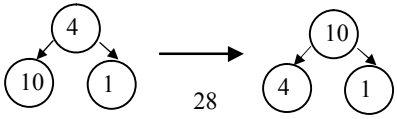


Рис. 6. Изменение первой пирамиды

вторая нижняя пирамида остается без изменения, третья и четвертая пирамиды изменяются:



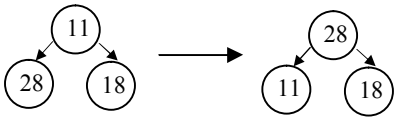


Рис. 7. Изменение других пирамид нижнего уровня

Далее рассматриваем пирамиды с узлами во 2-м и 3-м элементах:

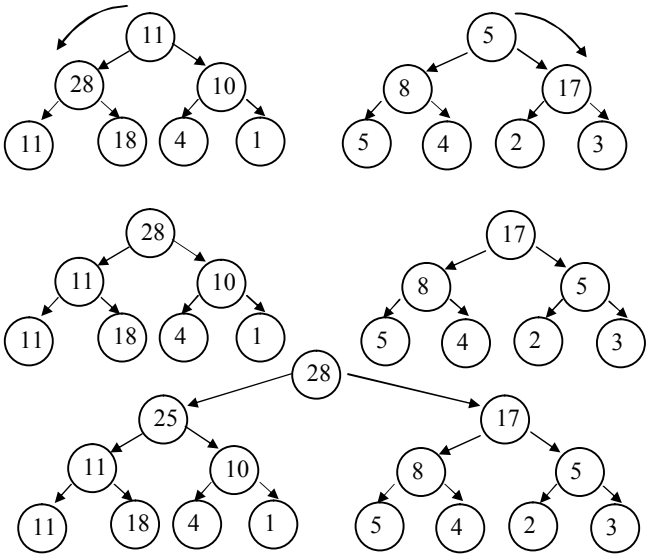


Рис. 8. Изменение пирамид второго и первого уровней

Очевидно, что процедура построения пирамиды должна зависеть от номера элемента, на котором строится пирамида, от количества элементов в массиве. Сам массив так же передается параметром.

В общем случае, для построения пирамиды с корнем в *L*-том можно описать следующий алгоритм.

Необходимо осуществить продвижение по дереву вниз:

- элемент с номером *L* меняется местами с большим из своих потомков;
- *L* изменяется на индекс большего потомка;
- алгоритм прекращает работу, когда элемент в позиции *L* больше своих потомков, или когда достигнут нижний уровень, т.е. *L >=N/2-1*.

Сортировка Хоара

Описание классического метода.

Сортировка Хоара использует принцип «разделяй и властвуй». Алгоритм делит весь массив на две части относительно среднего элемента, далее все элементы меньшие среднего элемента переносятся в левый список, а элементы большие среднего элемента в правый список. К двум полученным массивам применяется та же самая процедура. Таким образом, алгоритм является рекурсивным. Процедура сортировки (назовем ее *Qsort*) зависит от границ массива, к которым она применяется *Qsort(nF,nL)*.

Первый раз процедура вызывается для *nF = 0, nL = N-1*. Процедура ищет серединное значение массива *(nF+nL)/2*. Значение элемента с таким индексом обозначим *Med = x[(nF+nL)/2]*. Обозначим *i = nF, j = nL*. Пока не найден элемент левого подмассива, больший *Med*, увеличиваем *i*, далее просматриваем правый подмассив, уменьшая *j*, пока не найден элемент, меньший *Med*, если полученное значение *i* меньше или равно *j*, то обменяем найденные значения и перейдем к следующим значениям подмассивов. Увеличение *i* и уменьшение *j* выполняются до тех пор, пока *i<=j*. Далее, если *nF < j*, то вызывается *Qsort(nF,j)*, далее если *nL > i*, то вызывается *Qsort(i,nL)*.

Сортировка Хоара с выбором медианного элемента

Можно улучшить быструю сортировку, выбирая средний элемент таким образом, чтобы его значение было бы действительно близким к серединному значению массива.

Для этого можно пользоваться двумя стратегиями:

1. Выбор среднего значения осуществляется случайным образом (с использованием датчиков случайных чисел и информации о размерности массива). Т.к. разделяющий элемент выбирается при каждом вызове процедуры, случайный выбор может быть наиболее правильным и оградит от появления наихудшего случая – когда медианный элемент оказывается наименьшим или наибольшим.
2. Вторая стратегия состоит в случайном выборе 3-х элементов, по одному из начального, конечного и среднего интервалов сортируемого подмассива. Как разделяющий элемент используется среднее из этих трех чисел.

Сортировка Шелла

Сортировку Шелла называют еще сортировкой с уменьшающимся шагом. Метод Шелла сортирует элементы массива, отстоящие друг от друга на заданный интервал H_i. После того, как все элементы массива, отстоящие друг

от друга на H_i будут отсортированы, интервал H_i изменяется по правилу $H_{i+1} = (H_i - 1)/2$ (для массивов, содержащих более 500 элементов) и $H_{i+1} = (H_i - 1)/3$ (для массивов, содержащих менее 500 элементов). За H_0 принимается число элементов массива. Метод заканчивает работу, когда H_i становится меньше 1.

Внутри последовательности можно сортировать любым простым способом.

Пример работы сортировки на массиве:

9 2 3 1 11 12 4 6 9 8 7 10 5 .

$H_0 = 12, H_1 = (12-1)/3 = 3$. С таким шагом в массиве можно выделить три последовательности:

9 *2 3* ***1*** *11 12* ***4*** *6 9* ***8*** *7 10* ***5*** . После первого шага сортировки массив будет выглядеть следующим образом:

1 *2 3* ***4*** *6 9* ***5*** *7 10* ***8*** *11 12* ***9.*** $H_2 = (3-1)/3 = 0$. Но массив еще не сортировался с шагом 1, поэтому, следующий и последний шаг сортировки выполняется с шагом 1 :

1 2 3 4 5 6 7 8 9 10 11 12.

Комбинированная сортировка

Комбинация пузырька и сортировки Шелла. На каждом шаге сравниваются значения отстоящие друг от друга на заданное значение шага $H_{i+1} = 8 \cdot H_i / 11$, но такое сравнение происходит всего один раз. Как только значение смещения становится равным 1, выполняется сортировка до конца методом пузырька. За H_0 принимается число элементов массива.

Пример работы сортировки на массиве:

9 2 3 1 11 12 4 6 9 8 7 10 5 .

$H_0 = 12, H_1 = 8 \cdot H_0 / 11 = 8$. Можно выделить восемь последовательностей, три из них содержат по одному элементу:

9 *2* 3 **1** 11 *12* *4* *6* ***9*** *8* 7 **10** 5 .

После первого шага сортировки массив выглядит следующим образом:

9 *2* 3 **1** 5 *12* *4* *6* ***9*** *8* 7 **10** 11 . $H_2 = 8 \cdot 8 / 11 = 5$.

9 *2 3* 1 ***5*** ***12*** *4 6* 9 ***8*** *7 10 11*. После второго шага сортировки:

9 *2 3* 1 ***5*** *7 4 6* 9 ***8*** ***12*** *10 11*. $H_3 = 5 \cdot 8 / 11 = 3$.

9 *2 3* ***1*** *5 7* ***4*** *6* *9* ***8*** *12 10* ***11***. После третьего шага сортировки:

1 *2 3* ***4*** *5 7* ***8*** *6* ***9 9*** *12 10* ***11***. $H_4 = 3 \cdot 8 / 11 = 2$.

1 *2* ***3*** ***4*** ***5*** *7* ***8*** *6* ***9 9*** ***12*** *10* ***11***. После четвертого шага сортировки:

1 *2* ***3*** ***4*** ***5*** *6* ***8*** *7* ***9 9*** ***11*** *10* ***12.***

На пятом шаге сортировки выполняется алгоритм сортировки обменом *- 1 2 3 4 5 6 7 8 9 9 10 11 12.* Если при программировании учитывается правило досрочного выхода из внешнего цикла, то алгоритм закончится после выполнения второго внутреннего цикла.

Порядок выполнения работы

- получите вариант задания у преподавателя;
- Составьте алгоритм улучшенной сортировки.
- Реализуйте алгоритм на языке Си, добавив в программу подсчет количества сравнений и перестановок, проведенных алгоритмом.
- Выполните полученную программу на случайных массивах размерности 100, 1000, 10000.
- Сравните результаты работы простой и улучшенной сортировок по количеству сравнений и перестановок. Проанализируйте полученные результаты.

Поразрядная сортировка

Цель работы

Ознакомиться и реализовать алгоритмы поразрядной *LSD* и *MSD* сортировок.

Поразрядная сортировка

На практике сортировка применяется в основном к каким-либо структурам данных и выполняется по определенному ключу. Природа ключей может быть очень сложной. Совсем не обязательно, что на каждом шаге обрабатывается весь ключ.

Рассмотрим пример: известен автор – по трем первым буквам выбирается ящик каталога и в нем ведется поиск. Для того, чтобы сортировки были такими же эффективными будем рассматривать структуру ключей. Т.е. рассмотрим ключи как последовательности –

- Строки - последовательности символов
- Двоичные числа – последовательности битов
- Десятичные числа – последовательности десятичных разрядов.

Каждый элемент такой последовательности имеет строго определенный размер. Сортировки, основанные на обработке за раз одного такого элемента называются поразрядными *(radix)*.

Например, сортировка абонентов библиотеки – библиотекарь выставляет карточку клиента в отделение, на котором обозначена одна буква фамилии (всего отделений может быть 29). Если карточек много, то внутри отделения возможна сортировка по второй букве и т.д.. Это пример поразрядной сортировки с основанием 29.

Основа поразрядной сортировки – извлечь *i*- тый объект последовательности.

Существуют два базовых подхода: первый анализирует объекты слева направо (первыми обрабатываются наиболее значащие цифры). Такая сортировка называется *MSD* (*most significant digit*)- сортировкой.

Второй подход анализирует цифры справа налево (первыми обрабатываются меньшие разряды) – *LSD* (*last significant digit*).

Средства Си для реализации поразрядных сортировок

Для выделения *i* – того разряда десятичного числа *x* можно воспользоваться формулой $(x / R^i) \bmod R$. Например, извлечем разряд сотен числа 2875: **(2875/100) mod 10 →28 mod 10 ==8**

Для выделения символа строки используется обращение по индексу. Например, *char *x* – строка, *x[i]* – *i*-тый символ строки

При выделении *i*- того двоичного разряда можно использовать следующий алгоритм:

1. Выполнить сдвиг вправо на *i.*
2. Наложить на исходное число маску 1 (выполнить логическое умножение).
3. Полученное число вернуть в качестве результата.

Функция, выделяющая разряд может быть записана следующим образом:

```
// x -  исходное число, d – номер разряда
int digit(int x, int d)
{ int k = x>>d;
  k = k&1;
  return k;}
```

MSD - сортировка

Поразрядная сортировка

Предположим, следующие латинские буквы имеют коды:

a 000	**b 001**	**c 010**	**d 011**
e 100	**f 101**	**g 110**	**h 111**

Дана последовательность *e f d e c g h d e e*, необходимо упорядочить ее. Просмотрим последовательность слева направо, и найдем первый ключ, который начинается с бита 1, далее просмотрим последовательность справа налево и найдем ключ, начинающийся с бита 0. Обменяем их местами и будем выполнять этот процесс пока индексы просмотров не пересекутся:

100	011	011
101	**101**	010

011	011	011
100	100	100
010 →	**010** →	101
110	110	110
111	111	111
011	100	100
100	100	100
100	100	100

Первый шаг сортировки закончен. Получено два подмассива: с первой единицей и с первым нулем.

Рекурсивно применим ту же самую процедуру к полученным массивам по следующему разряду:

011	
010	
011	
100	
101	Без изменений, все элементы разряда равны 1.
110	
111	
100	
100	
100	

По третьему разряду:

011	010	
010	011	
011	011	
100	100	
101	101	Сортировка первого подмасссива законче-
110	110	на, так как рассмотрен последний разряд
111	111	
100	100	
100	100	
100	100	

Начнем сортировать по второму разряду второй подмассив:

011	010	010
010	011	011
011	011	011
100	100	100

101	101	101
110	100	100
111	**111**	100
100	100	100
100	**100**	111
100	110	110

Сортировка по третьему разряду двух полученных подмассивов:

010	010	010
011	011	011
011	011	011
100	100	100
101	100	100
100	100	100
100	100	100
100	101	101
111	**111**	**110**
110	**110**	**111**

Сортировка закончена, получен отсортированный массив:
c d d e e e ef g h.

Алгоритм сортировки в общем виде можно записать следующим образом:

```
Radix_bin(X,F,L,d)
{
        Если d<0 или F==L то выход
        I=F, J = L;
        1: пока I<J
        если байт d в x[I] == 0 то I++
        пока I<J
       если байт d в x[j] == 1 то J++
        Если I<J то меняем местами x[I] и x[J]
        возврат на 1.
        radix_bin(X,F,J,d-1)
        radix_bin(X,I,L,d-1)
  }
```

Поразрядная MSD сортировка

Если в быстрой двоичной сортировке разделение происходит на два подмассива (0 и 1), то *MSD* сортировка обобщает понятие поразрядной сортировки по произвольному основанию *R* – происходит разделение всего массива на *R* подмассивов.

Таким образом, в функции будет выполнено *R* рекурсивных вызовов.

Для больших значений *R* можно использовать следующую схему передвижения элементов к своему подмассиву. Создаются два вспомогательных

массива – массив счетчиков для каждого из значений *R* и временный массив для хранения передвинутых элементов. Просматривается исходный массив первый раз и подсчитывается сколько раз встретилась буква "*а"* среди первых букв слов, буква *"б"* и т.д.. При втором проходе элементы из исходного массива записываются во временный, используя те точки разделения, которые получены при первом проходе. Для вычисления точек разделения можно использовать следующий алгоритм:

3 4 2 7 9 → 3 7 9 16 25

Каждый элемент массива сложим с предыдущим. Далее просмотрим основной массив – если первая буква *"а"*, то ставим ее на *2 (3-1)* место и первый элемент массива разделений уменьшаем: *2 7 9 16 25* и т.д..

Основная часть работы происходит на первом же этапе разделения. Можно улучшить алгоритм *MSD* если для сортировки подмассивов маленькой размерности использовать алгоритм простой сортировки.

Поразрядная LSD сортировка

Сортировка работает только в случае устойчивого способа перестановки элементов. К устойчивым сортировкам относится сортировка вставками, поэтому можно применить его. Рассмотрим на примере:

01001	10010	10100	00000	00000	00000	0
10010	10110	00000	10000	10000	00011	3
11011	10100	10000	01001	10010	01001	9
10110	00000	01001	10010	00011	10000	16
00011	10000	10010	11011	10100	10010	18
10100	01001	10110	00011	10110	10100	20
00000	11011	11011	10100	01001	10110	22
10000	00011	00011	10110	11011	11011	27

Сортировка нерекурсивная.

На основании *R* выполняется аналогичным образом и остается такой же нерекурсивной.

Порядок выполнения

- Получите вариант задания у преподавателя.
- Составьте алгоритм поразрядной сортировки.
- Реализуйте алгоритм на языке Си.
- Проанализируйте полученные результаты.

Рекомендуемая литература

1. Павловская Т. А. C/C++. Программирование на языке высокого уровня : Учебник для вузов / Т. А. Павловская. - СПб. : Питер, 2007. - 460 с
2. Новиков Ф. А. Дискретная математика для программистов : Учебное пособие для вузов / Ф. А. Новиков. - СПб. : Питер, 2003. - 302[2] с.
3. Вирт Н. Алгоритмы и структуры данных. / Н. Вирт– Спб: Н. Диалект, 2001 – 250с.
4. Хаггарти Р. Дискретная математика для программистов : Пер. с англ. : Учебное пособие для вузов / Р. Хаггарти ; ред. пер. С. А. Кулешов, доп. А. А. Ковалев, доп. В. А. Головешкин, доп. М. В. Ульянов. - 2-е изд., доп. - М. : Техносфера, 2005. - 399[1] с.
5. Иванов Б. Н. Дискретная математика. Алгоритмы и программы : Учебное пособие для вузов / Б. Н. Иванов. - М. : Лаборатория Базовых Знаний, 2003. - 288 с.
6. Пермякова Н.В. Программирование на языке высокого уровня : учебное пособие: в 2 ч. / Н. В. Пермякова ; Федеральное агентство по образованию, Томский государственный университет систем управления и радиоэлектроники, Кафедра автоматизации обработки информации. Ч.1. - Томск : ТМЦДО, 2007 – 195 с.