государственное бюджетное профессиональное образовательное учреждение
«Пермский химико-технологический техникум»

МЕТОДИЧЕСКИЕ УКАЗАНИЯ ДЛЯ ОБУЧАЮЩИХСЯ ПО ВЫПОЛНЕНИЮ ПРАКТИЧЕСКИХ РАБОТ

для специальности 09.02.03 Программирование в компьютерных системах
по дисциплине ОП.08 «Теория алгоритмов»

2014

СОДЕРЖАНИЕ

ВВЕДЕНИЕ 3
ПРАВИЛА ВЫПОЛНЕНИЯ ПРАКТИЧЕСКИХ РАБОТ 5
ОПИСАНИЕ РАБОЧЕГО МЕСТА ОБУЧАЮЩЕГОСЯ 6
ПРАКТИЧЕСКИЕ РАБОТЫ 7
Практическая работа № 1 7
Практическая работа № 2 15
Практическая работа № 3 20
Практическая работа № 4 24
Практическая работа № 5 27
Практическая работа № 6 31
Практическая работа № 7 36
Практическая работа № 8 38
Практическая работа № 9 39
Практическая работа № 10 40
Практическая работа № 11 43
Практическая работа № 12 48
Практическая работа № 13 53
Практическая работа № 14 64

ВВЕДЕНИЕ

Учебная дисциплина ОП.08 «Теория алгоритмов» является обязательной в цикле общепрофессиональных дисциплин основной профессиональной образовательной программы по специальности 09.02.03 «Программирование в компьютерных системах».

В результате освоения учебной дисциплины обучающийся должен уметь:

- разрабатывать алгоритмы для конкретных задач;
- определять сложность работы алгоритмов;

В результате освоения учебной дисциплины обучающийся должен знать:

- основные модели алгоритмов;
- методы построения алгоритмов;
- методы вычисления сложности работы алгоритмов

Формируемые дисциплиной компетенции:

ПК 1.1. Выполнять разработку спецификаций отдельных компонент.

ПК 1.2. Осуществлять разработку кода программного продукта на основе готовых спецификаций на уровне модуля.

ОК 1. Понимать сущность и социальную значимость своей будущей профессии, проявлять к ней устойчивый интерес.

ОК 2. Организовывать собственную деятельность, выбирать типовые методы и способы выполнения профессиональных задач, оценивать их эффективность и качество.

ОК 3. Принимать решения в стандартных и нестандартных ситуациях и нести за них ответственность.

ОК 4. Осуществлять поиск и использование информации, необходимой для эффективного выполнения профессиональных задач, профессионального и личностного развития.

ОК 5. Использовать информационно-коммуникационные технологии в профессиональной деятельности.

ОК 6. Работать в коллективе и в команде, эффективно общаться с коллегами, руководством, потребителями.

ОК 7. Брать на себя ответственность за работу членов команды (подчиненных), за результат выполнения заданий.

ОК 8. Самостоятельно определять задачи профессионального и личностного развития, заниматься самообразованием, осознанно планировать повышение квалификации.

ОК 9. Ориентироваться в условиях частой смены технологий в профессиональной деятельности.

Методические указания предназначены для проведения практических занятий по дисциплину «Теория алгоритмов», закрепления теоретических знаний и получения навыков работы в области прикладного программирования.

Методические указания разработаны в соответствии с рабочей программой дисциплинs «Теория алгоритмов» по специальности 09.02.03 «Программирование в компьютерных системах».

Содержание методических указаний по выполнению практических работ соответствует требованиям Федерального государственного стандарта среднего профессионального образования по специальности 09.02.03. «Программирование в компьютерных системах».

По учебному плану, и в соответствии с рабочей программой профессионального дисциплины «Теория алгоритмов», на изучение обучающимися предусмотрено 84 часа, в том числе: обязательной аудиторной учебной нагрузки 56 часов; 28 часов практических занятий, самостоятельной работы обучающегося 28 часов.

Методические указания включают 14 практических работ. Каждая практическая работа содержит сведения о теме, цели ее проведения и формируемых компетенциях,

включает пояснения к работе, содержание отчета, контрольные задания или вопросы, список литературы.

К выполнению практических работ обучаемые приступают после подробного изучения соответствующего теоретического материала и прохождения инструктажа по технике безопасности.

Характер практических работ репродуктивный и частично-репродуктивный.

Примечание:

Репродуктивный характер – обучающиеся пользуются подробными инструкциями, в которых указаны: цель работы, пояснения, оборудование, аппаратура, материалы и их характеристики, порядок выполнения работы, таблицы, выводы (без формулировок), контрольные вопросы, литература.

Частично-поисковый характер – обучающиеся не пользуются подробными инструкциями, им не задан порядок выполнения заданий, от студентов требуется самостоятельный подбор оборудования, выбор способов выполнения работы, справочной литературы.

Поисковый характер – обучающиеся должны решить новую для них проблему, опираясь на имеющиеся у них теоретические знания.

Работы частично-поискового и поискового характера выполняются, как правило, при изучении дисциплин, составляющих ядро конкретной специальности, а так же дисциплин, связанных с обслуживанием, эксплуатацией и ремонтом различного оборудования.

ПРАВИЛА ВЫПОЛНЕНИЯ ПРАКТИЧЕСКИХ РАБОТ

1. Студент должен прийти на практическое занятие, подготовленным к выполнению работы. Студент, не подготовленный к работе, не может быть допущен к ее выполнению.
2. Каждый студент выполняет вариант задания, указанного преподавателем в начале занятия.
3. Каждый студент после выполнения работы должен представить отчет о проделанной работе.
4. Таблицы и схемы следует выполнять с помощью чертежных инструментов (линейки, циркуля и т. д.) карандашом с соблюдением ЕСПД.
5. В заголовках граф таблиц обязательно проводить буквенные обозначения величин и единицы измерения в соответствии с ЕСПД.
6. Вспомогательные расчеты можно выполнить на отдельных листах, а при необходимости на листах отчета.
7. Если студент не выполнил практическую работу или часть работы, то он может выполнить работу или оставшуюся часть во внеурочное время, согласованное с преподавателем.
8. Оценку по практической работе студент получает, с учетом срока выполнения работы, если:
    1. работа выполнена правильно и в полном объеме;
    2. студент может пояснить выполнение любого этапа работы;
    3. отчет выполнен в соответствии с требованиями к выполнению работы;
    4. студент ответил на дополнительные теоретические вопросы преподавателя.
9. Для получения допуска к экзамену студент должен выполнить все предусмотренной программой работы после сдачи отчетов при удовлетворительных оценках.

После окончания занятий обучающиеся приводят в порядок рабочее место, показывают преподавателю полученные результаты. После утверждения преподавателем предъявленных результатов, каждый обучающийся оформляет отчет о проделанной работе, представляет его на проверку и подпись преподавателю в тот же день или на следующем практическом занятии.

ОПИСАНИЕ РАБОЧЕГО МЕСТА ОБУЧАЮЩЕГОСЯ

1. Практические работы по дисциплине выполняются в аудитории, компьютерном классе.
2. Для выполнения практических работ необходимы:
   - методические указания;
   - чертежные инструменты;
   - конспект лекций и учебник;
   - персональный компьютер.
   - операционная система Windows;
   - приложения MS Office;
   - Паскаль АВС;
   - эмулятор исполнителя Машина Поста;
   - эмулятор исполнителя Машина Тьюринга
   - эмулятор исполнителя Нормальные алгорифмы Маркова

3. Выполнение расчетов и оформление отчета студент выполняет индивидуально, согласно указанного преподавателем варианта.
4. По выполнению всех этапов задания и их оформления студент представляет отчет по практической работе.
5. Для получения оценки студент защищает практическую работу по предоставленному отчету и отвечая на вопросы преподавателя по теории.

ПРАКТИЧЕСКИЕ РАБОТЫ

Практическая работа № 1

Тема: Реализация алгоритмов на машине Поста.

Цель: закрепление умений по реализации алгоритмов на машине Поста.

Пояснения к работе:

Машина Поста — это абстрактная (т.е. не существующая в арсенале действующей техники), но очень простая вычислительная машина. Она способна выполнять лишь самые элементарные действия, и потому ее описание и составление простейших программ может быть доступно ученикам начальной школы. Тем не менее на машине Поста можно запрограммировать — в известном смысле — любые алгоритмы. Изучение машины Поста можно рассматривать как начальный этап обучения теории алгоритмов и программированию.

Машина Поста состоит из *ленты* и *каретки* (называемой также *считывающей* и*записывающей головкой*). Лента бесконечна и разделена на секции одинакового размера — *ячейки*.

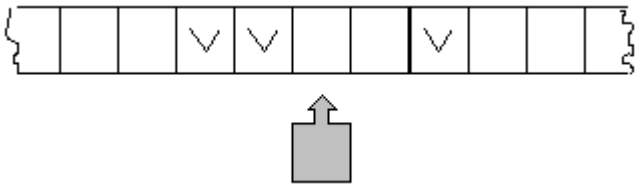


Рис. 1. В каждый момент времени каретка указывает на одну из ячеек

В каждой ячейке ленты может быть либо ничего не записано, либо стоять метка**V**. Информация о том, какие ячейки пусты, а какие содержат метки, образует*состояние ленты*. Иными словами, состояние ленты — это распределение меток по ячейкам. Состояние ленты меняется в процессе работы машины. Заметим, что наличие метки в ячейке можно интерпретировать как “1”, а отсутствие — “0”. Такое двоичное представление информации подобно представлению, используемому практически во всех современных ЭВМ.

Каретка может передвигаться вдоль ленты влево и вправо. Когда она неподвижна, она стоит против ровно одной ячейки ленты; говорят, что каретка обозревает одну ячейку. За единицу времени каретка может совершить одно из трех действий: стереть метку, поставить метку, совершить движение на соседнюю ячейку.*Состояние машины Поста* складывается из состояния ленты и положения каретки.

Действия каретки подчинены программе, состоящей из перенумерованного набора команд (команды можно представлять как строки программы). Команды бывают шести типов:

1. записать 1 (метку), перейти к *i*-й строке программы;
2. записать 0 (стереть метку), перейти к *i*-й строке программы;
3. сдвиг влево, перейти к *i*-й строке программы;
4. сдвиг вправо, перейти к *i*-й строке программы;
5. останов;
6. если 0, то перейти к *i*, иначе перейти к *j*.

Приведем список недопустимых действий, ведущих к аварийной остановке машины:

- попытка записать 1 (метку) в заполненную ячейку;
- попытка стереть метку в пустой ячейке;
- бесконечное выполнение (вообще говоря, это трудно назвать аварийным остановом, но бессмысленное повторение одних и тех же действий — зацикливание — ничуть не лучше вышеперечисленного).

Машина Поста, несмотря на внешнюю простоту, может производить различные вычисления, для чего надо задать начальное состояние каретки и программу, которая эти вычисления сделает. Машиной эта математическая конструкция названа потому, что при ее построении используются некоторые понятия реальных машин (ячейка памяти, команда и др.). Условимся каждый шаг программы обозначать номером. Команды машины будем обозначать следующим образом:

→	Шаг вправо
←	Шаг влево
V	Записать отметку
X	Стереть отметку
? *a*; *b*	Просмотреть ячейку; если в ячейке находится 0, то перейти на команду с номером *a*, иначе на команду с номером *b*
!	Останов

Будем говорить, что мы можем *применить программу к текущему состоянию машины Поста*, если выполнение программы не приведет к зацикливанию, т.е. рано или поздно мы выполним команду *останов*.

Пример программы, которая не применима ни к одному состоянию машины Поста:

1. → 1

2. !

Рассмотрим задачу для машины Поста и ее решение.

Задача. На ленте проставлена метка в одной-единственной ячейке. Каретка стоит на некотором расстоянии левее этой ячейки. Необходимо подвести каретку к ячейке, стереть метку и остановить каретку слева от этой ячейки.

Решение. Сначала попробуем описать алгоритм обычным языком. Поскольку нам известно, что каретка стоит напротив пустой ячейки, но неизвестно, сколько шагов нужно совершить до пустой ячейки, мы можем сразу сделать шаг вправо; проверить, заполнена ли ячейка; если она пустая, то повторять эти действия до тех пор, пока не наткнемся на заполненную ячейку. Как только мы ее найдем, мы выполним операцию стирания, после чего нужно будет лишь сместить каретку влево и остановить выполнение программы.

Программа для машины Поста:

1. → 2

2. ? 1; 3

3. X 4

4. ← 5

5. !

Задание:

Пояснения к условиям задач

1) В задачах под *массивом* понимается последовательность подряд идущих меток, ограниченная пустыми ячейками.

2) Если в задаче говорится, что на ленте задано число в унарной системе, то имеется в виду, что натуральное число *n* закодировано с помощью массива длины *n*.

3) В задачах при описании начального состояния ленты будем указывать то, что записано начиная с самой левой непустой ячейки и заканчивая самой правой непустой ячейкой. При этом будем использовать следующие обозначения: *n* подряд идущих меток будем обозначать 1*n*, а *m* пустых ячеек — 0*m*. При обозначении одной заполненной или пустой ячейки будем писать просто 1 или 0, соответственно.

К примеру, запись “12012” будет соответствовать записи “11011” на ленте.

4) Если не сказано ничего о местонахождении каретки в начальный момент времени, то будем считать, что каретка обозревает ячейку с самой левой меткой.

1. Применимость программ. Определение результата выполнения программ

1. Выяснить, применимы ли программы к заданным состояниям машины Поста, указать результат работы машины Поста для каждого состояния.

a)

1. ? 3; 2	6. → 7	Начальное состояние ленты:
2. → 1	7. ? 8; 9	1) $1^{3}0^{2}1^{2}$
3. → 4	8. !	2) $1^{3}01^{3}$
4. ? 6; 5	9. → 4	3) $10\,[01]^{2}1$
5. ← 1		

b)

1. ? 4; 2	7. V 8	Начальное состояние ленты:
2. X 3	8. ← 9	1) $1^{4}01$
3. → 9	9. ? 11; 10	2) $1^{3}01^{2}$
4. V 5	10. → 1	3) 1^{6}
5. → 6	11. !	
6. ? 7; 6		

c)

1. ? 4; 2	7. ← 8	Начальное состояние ленты:
2. X 3	8. ? 9; 11	1) 101^{2}
3. → 6	9. V 10	2) $1^{2}0^{2}1$
4. V 5	10. ← 1	3) $[10]^{2}1$
5. → 1	11. !	
6. ? 4; 7		

Ответы:

a) 1) 1110011000
   2) зацикливание
   3) 1001011000

b) 1) зацикливание
   2) 010011
   3) 01010110

c) 1) зацикливание (…111)
   2) зацикливание (…1111001)
   3) зацикливание (1010111…)

2. Определить состояние, в котором окажется машина Поста в результате выполнения программы при заданном начальном состоянии ленты.

Пояснение: выделенная цифра, например ***1***, означает, что эту ячейку каретка обозревает в начальный момент времени.

a)

1. ? 2; 4	5. → 6	Начальное состояние ленты:
2. V 3	6. ? 8; 7	1) $1^{2}\mathbf{1}1^{2}01$
3. !	7. ← 6	2) $1^{2}\mathbf{1}1^{3}$
4. X 5	8. → 1	

b)

1. ? 2; 3	8. ← 2	Начальное состояние ленты:
2. !	9. ? 12; 10	1) $1\mathbf{1}1^{4}$
3. → 4	10. X 11	2) $\mathbf{1}1^{2}01$
4. ? 7; 5	11. → 1	3) $10\mathbf{1}1^{3}$
5. X 6	12. V 13	
6. → 9	13. ← 1	
7. V 8		

Решение. Выделенная цифра показывает, на какой ячейке остановится машина.

a) 1) 11000000***1***
   2) 1100000***1***

b) 1) 11001***0***1
   2) 100***0***1
   3) ***1***11111

3. Написать программы для машины Поста, которые обладают следующими свойствами:

- программа применима к любому состоянию машины Поста;
- программа не применима ни к какому состоянию машины Поста, и зона работы для любого начального состояния — бесконечная;

- программа не применима ни к какому состоянию машины Поста, и зона работы для любого начального состояния ограничена одним и тем же числом ячеек, не зависящим от выбранного начального состояния ленты;
- программа применима к состояниям 1^{3n} ($n \geq 1$) и не применима к состояниям 1^{3n+a}, где a = 1, 2 и $n \geq 1$;
- программа применима к состояниям 1^a01^a, где $a \geq 1$, и не применима к 1^a01^b, $a \neq b$ (a и $b \geq 1$).

Решение

- программа, применимая к любому состоянию машины Поста:

  ! 1
- программа, не применимая ни к какому состоянию машины Поста, и зона работы для любого начального состояния бесконечна:

  –> 1
- машина, не применимая ни к какому состоянию машины Поста, и зона работы для любого начального состояния ограничена одним и тем же числом ячеек, не зависящим от выбранного начального состояния ленты:

  1. –> 2
  2. <– 1
- программа, применимая к состояниям 1^{3n}, и не применимая к состояниям 1^{3n+a}, где a = 1, 2 и $n \geq 1$:

1. ? 8, 2	5. ? 7, 6
2. 3	6. 3
3. ? 7, 4	7. 7
4. 5	8. !

- программа применима к состояниям 1^a01^a, где $a \geq 1$, и не применима к 1^a01^b, $a \neq b$ ($a \geq 1$ и $b \geq 1$):

  в качестве примера такой программы может быть взята программа, удаляющая последовательно по одному элементу из каждого из двух массивов меток и уходящая на бесконечность в случае, если остались элементы в одном из массивов.

2. Арифметические задачи

Программы для решения всех задач этого раздела могут быть интерпретированы как выполнение элементарных арифметических операций. Важно показать, как с помощью простейших операций, которыми располагает машина Поста, можно выполнять арифметические операции — основу любого современного процессора.

4. На ленте задан массив меток. Увеличить длину массива на 2 метки. Каретка находится либо слева от массива, либо над одной из ячеек самого массива.

Решение.

1. ? 2; 3 (команды 1 и 2 — передвигаем каретку к массиву)
2. –> 1
3. –> 4 (команды 3 и 4 — передвигаем каретку к концу массива)
4. ? 5; 3
5. V 6 (команды 5–7 — ставим 2 метки в конце массива)
6. –> 7
7. V 8
8. !

5. Даны два массива меток, которые находятся на некотором расстоянии друг от друга. Требуется соединить их в один массив. Каретка находится над крайней левой меткой первого массива.

Решение.

1. X 2	6. ? 8; 7
2. → 3	7. !
3. ? 4; 2	8. ← 9
4. V 5	9. ? 10; 8
5. → 6	10. → 1

6. На ленте задана последовательность массивов, включающая в себя один и более массивов. При этом два соседних массива отделены друг от друга одной пустой ячейкой. Необходимо на

ленте оставить один массив длиной равной сумме длин массивов, присутствовавших изначально. Каретка находится над крайней левой меткой первого (левого) массива.

Решение.

1. X 2	6. ? 10; 7
2. → 3	7. ← 8
3. ? 4; 2	8. ? 9; 7
4. V 5	9. → 1
5. → 6	10. !

7. На ленте заданы два массива — *m* и *n*, *m* > *n*. Вычислить разность этих массивов. Каретка располагается над левой ячейкой правого массива.

Решение. Запишем решение алгоритма в словесной форме.

1. Ищем правый край массива *m*, двигаясь слева направо.
2. Стираем правую метку массива *m*.
3. Ищем правый край массива *n*, двигаясь слева направо.
4. Стираем левую метку массива *n*.
5. Проверяем, мы стерли последнюю метку в массиве *n* (в этом случае следующая справа ячейка должна быть пустой)?
6. Если стерли последнюю метку, то конец алгоритма.
7. Иначе ищем правый конец массива *m*, двигаясь справа налево.
8. Переход на шаг 2.

1. –> 2 (команды 1–3: ищем левую метку массива *m*)
2. ? 3; 1
3. <– 4
4. X 5 (стираем левую метку массива *m*)
5. ? 6; 7
6. –> 5
7. X 8 (стираем левую метку массива *n*)
8. –> 9
9. ? 12; 10 (стерли последнюю метку в массиве *n*?)
10. <– 11 (ищем левый край массива *m*)
11. ? 10; 4
12. !

8. На ленте заданы два массива. Найти модуль разности длин массивов. Каретка располагается над первой ячейкой левого массива.

Решение.

1. –> 2
2. ? 3; 1 (идем до конца первого массива)
3. <– 4
4. X 5 (удаляем крайний правый элемент 1-го массива)
5. <– 6
6. ? 14; 7 (проверяем, что в 1-м массиве еще остались метки)
7. –> 8
8. ? 7; 9
9. X 10 (удаляем первую метку 2-го массива)
10. –> 11
11. ? 17; 12 (проверяем, что во 2-м массиве еще остались метки, иначе — завершение)
12. <– 13
13. ? 12; 4
14. –> 15 (мы удалили полностью 1-й массив)
15. ? 14; 16
16. X 17
17. !

9. На ленте задан массив. Удвоить массив в два раза. Каретка располагается над первой ячейкой массива.

Решение. В результате работы программы справа от исходного массива будет сформирован новый массив удвоенной длины, исходный массив будет стерт.

1. ← 2	9. V 10
2. ? 3; 1	10. → 11
3. → 4	11. V 12
4. X 5	12. ← 13
5. → 6	13. ? 14; 12
6. ? 7; 5	14. ← 15
7. → 8	15. ? 16; 1
8. ? 9; 7	16. ! 16

10. На ленте задан массив. Вычислить остаток от деления длины заданного массива на 3. Каретка располагается над первой ячейкой массива.

Решение.

1. ? 2; 3	8. → 9
2. !	9. ? 10; 12
3. X 4	10. V 11
4. → 5	11. ← 6
5. ? 6; 7	12. X 13
6. V 2	13. → 1
7. X 8	

11. На ленте машины Поста расположен массив из *n* меток. Составить программу, действуя по которой машина выяснит, делится ли число *n* на 3. Если да, то после массива через одну пустую ячейку поставить метку.

Решение. Нужно проверить, что массив состоит не менее чем из трех меток, сместиться правее них и снова решать ту же задачу. Если правее очередных трех меток окажется пробел, то за ним поставить еще одну метку.

1. → 2	6. → 7
2. ? 3; 4	7. ? 8; 1
3. !	8. → 9
4. → 5	9. V 3
5. ? 3; 6	

3. Ориентация на ленте

12 На ленте имеется некоторое множество меток (общее количество меток не менее 1). Между метками множества могут быть пропуски, длина которых составляет одну ячейку. Заполнить все пропуски метками.

Решение.

1. → 2	5. !
2. ? 3, 1	6. ← 7
3. → 4	7. V 1
4. ? 5, 6	

13. На ленте имеется массив из *n* отмеченных ячеек. Каретка обозревает крайнюю левую метку. Справа от данного массива на расстоянии в *m* ячеек находится еще одна метка. Составьте для машины Поста программу, придвигающую данный массив к данной ячейке.

Решение.

1. X 2 (удаляем левую метку массива)
2. –> 3
3. ? 4; 2 (передвигаем каретку к концу массива)
4. V 5 (ставим справа от массива метку, раннее нами была удалена самая левая метка)
5. –> 6
6. ? 7; 10 (проверяем, передвинули ли мы уже наш массив к заданной метке)
7. <– 8
8. ? 9; 7 (идем к левой метке массива)
9. –> 1 (и начинаем все сначала)
10. !

14. Известно, что на ленте машины Поста находится метка. Напишите программу, которая находит ее.

Решение. Этот алгоритм решения заимствован из замечательной книги В.А. Успенского “Машина Поста”. Мы не знаем, в какую сторону нам надо двигаться, но, в какую бы сторону мы ни пошли, может случиться, что метка стоит в другой стороне. Очевидно, что нам надо двигаться попеременно, то в одну сторону, то в другую, постоянно увеличивая размах своих колебаний. Но как определить момент, когда надо поворачивать, т.е. менять направление? Выход из положения

есть. Вначале работы выставим метки слева и справа от исходного положения каретки, а затем будем ходить между ними и передвигать их.

1. V 2 (выставили левую метку)
2. –> 3
3. ? 5; 4
4. ! (нашли метку, конец)
5. V 6 (выставили правую метку)
6. <– 7 (ищем левую метку)
7. ? 6; 8
8. X 9 (стираем левую метку)
9. <– 10
10. ? 11; 4
11. V 12 (передвигаем левую метку)
12. –> 13 (ищем правую метку)
13. ? 12; 14
14. X 15 (стираем правую метку)
15. –> 3 (повторяем действия)

4. Действия над заданным на ленте множеством меток

15. Дан массив меток. Каретка располагается где-то над массивом, но не над крайними метками. Стереть все метки, кроме крайних, и поставить каретку в исходное положение.

Решение. Метку, которую мы обозреваем в начальный момент времени, мы сотрем самой последней, т.к. нам нужно будет вернуть каретку в начальное положение. Мы можем, к примеру, сначала стереть все метки массива, кроме крайней справа от исходного положения, затем стереть все метки, кроме крайней слева от исходного положения. Потом вернуться к оставленной нами в самом начале метке.

1. –> 2
2. X 3
3. –> 4
4. ? 5, 2 (удаляем метки справа от исходного положения)
5. <– 6
6. V 7
7. <– 8 (возвращаемся к исходному положению)
8. ? 7; 9
9. <– 10
10. X 11
11. <– 12
12. ? 13; 10 (удаляем метки слева от исходного положения)
13. –> 14
14. V 15
15. –> 16
16. ? 15; 17 (возвращаемся к исходному положению)
17. X 18 (удаляем метку, соответствующую исходному положению каретки)
18. !

16. На ленте машины Поста расположен массив из *n* меток (метки расположены через пробел). Нужно сжать массив так, чтобы все *n* меток занимали *n*расположенных подряд ячеек.

Решение. Идея решения состоит в последовательном придвижении каждой отдельной метки к уже сформированному массиву. Считаем, что каретка находится над левой меткой массива. Программа решения данной задачи эквивалентна программе сложения произвольного количества чисел (см. задачу 6).

17. Дано несколько массивов меток. Удалить четные массивы. Каретка находится над первым массивом.

Решение.

1. –> 2
2. ? 3; 1 (идем до конца нечетного массива)
3. –> 4
4. ? 5; 6 (смотрим, есть ли еще массивы)
5. ! (массивов больше нет — завершение)
6. X 7 (удаляем четный массив)
7. –> 8

8. ? 9; 6
9. –> 10
10. ? 5; 1 (смотрим: есть ли еще массивы)

18. На ленте машины Поста расположено n массивов меток, отделенных друг от друга свободной ячейкой. Каретка находится над крайней левой меткой первого массива. Определить количество массивов.

Решение. Идея решения такова: будем “считать” массивы слева направо, удаляя каждый “посчитанный” массив. При этом слева от последовательности оставшихся массивов будем держать массив меток, длина которого соответствует числу “посчитанных” массивов.

1. → 2	8. ← 9
2. ? 5, 3	9. ? 8, 10
3. X 4	10. → 11
4. → 2	11. V 12
5. → 6	12. → 13
6. ? 7, 8	13. ? 12, 2
7. !	

19. На ленте машины Поста расположен массив из $2n - 1$ меток. Составить программу удаления средней метки массива.

Решение. Идея решения состоит в следующем: во вторых ячейках от каждого края массива ставим “маячки-пузырьки” (эти ячейки делаем пустыми). Далее последовательно перемещаем к центру левый и правый пузырьки. Эти пузырьки встретятся ровно на центральном элементе исходного массива. При реализации программы надо отдельно учесть три случая: $n = 1$, $n = 3$, $n > 3$. Считаем, что в начале работы каретка стоит на самой левой метке массива.

1. –> 2
2. ? 3; 4
3. <– 4 ($n = 1$)
4. X 5
5.
6. –> 7
7. ? 8; 6
8. 9
9. <– 10
10. ? 20; 11
11. X 12 ($n > 3$)
12. <– 13
13. ? 14; 12
14. V 15 (дошли до левого конца)
15. –> 16
16. X 17
17. –> 18
18. ? 19; 17
19. V 9 (дошли до правого конца)
20. ! (стерли центральную метку, конец)

20. На ленте машины Поста расположен массив из $2n$ ячеек. Составить программу, по которой машина Поста раздвинет на расстояние в одну ячейку две половины данного массива.

Решение. Идея решения состоит в следующем. Сначала между двумя левыми и двумя правыми метками ставим “маячки” — пустые клетки. Первым ставим левый маячок. Затем поочередно сдвигаем эти маячки к центру. Как только маячки сомкнутся, вместо правого маячка ставим метку, идем к правому краю массива и удаляем самую правую метку. Для простоты решения считаем, что каретка стоит под самой левой меткой.

21. Написать программу, которая осуществляет преобразование 1^n01^m –> 1^m01^n($n \geq 1$ и $m \geq 1$).

Решение. Правый массив длины m остается на месте, левый массив переносится слева направо относительно неподвижного массива.

1. X 2	10. ? 11; 9
2. → 3	11. ← 12
3. ? 4; 2	12. ? 13; 11
4. → 5	13. ← 14
5. ? 6, 4	14. ? 15, 16
6. → 7	15. ! 15
7. ? 8; 6	16. ← 17
8. V 9	17. ? 18; 16
9. ← 10	18. → 1

5. Сравнение

22. На ленте расположены два массива разной длины. Каретка обозревает крайний элемент одного из них. Составьте программу для машины Поста, сравнивающую длины массивов и стирающую больший из них. Отдельно продумайте случай, когда длины массивов равны.

Решение аналогично нахождению разности двух чисел.

23. На ленте машины Поста находятся два массива в m и n меток. Составить программу выяснения, одинаковы ли массивы по длине.

Решение аналогично нахождению разности двух чисел.

24. Дано N массивов меток. Массивы разделены тремя пустыми ячейками. Количество меток в массиве не меньше двух. Если количество меток в массиве кратно трем, то стереть метки в этом массиве через одну, в противном случае стереть весь массив. Каретка находится над крайней левой меткой первого массива.

Решение. В задаче присутствует большое количество условий. Вместе с тем реализация этих условий требует лишь внимательного составления программы.

Содержание отчета

– Название работы;
– Цель работы;
– Формулировку задания;
– Решенные задания, с пояснением решения в виде таблиц, рассуждений, рисунков.

Контрольные вопросы

1. Перечислите систему команд в машине Поста.
2. Какие свойства алгоритма демонстрирует машина Поста?

Список литературы:

Основная литература

1. Окулов С. М. Программирование в алгоритмах. / М. Окулов. – М.: БИНОМ. Лаборатория знаний, 2002. – 341 с.
2. Иванов Б. Н. Дискретная математика. Алгоритмы и программы. – М.: Лаборатория Базовых Знаний, 2002.

Дополнительная литература

1. Семакин И. Г. Основы программирования: учебник для сред. проф. образования / И. Г. Семакин, А. П. Шестаков. – М.: Издательский центр «Академия», 2007.
2. Акимов О.Е. Дискретная математика: логика, группы, графы. – М.: Лаборатория Базовых Знаний, 2001.

Интернет ресурсы

1. *http://math.ru/lib/plm/54* (электронная версия книги В.А. Успенского “Машина Поста”).
2. *http://softsearch.ru/programs/45-346-interpretator-mashiny-posta-download.shtml*(имитатор машины Поста).

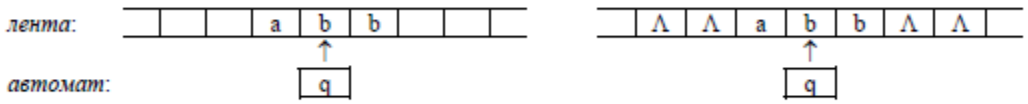
Практическая работа № 2

Тема: Реализация алгоритмов на машине Тьюринга.

Цель: закрепление умений по реализации алгоритмов на машине Тьюринга.

Пояснения к работе:

Структура машины Тьюринга *Машина Тьюринга (МТ) состоит из двух частей* - ленты *и* автомата



Лента используется для хранения информации. Она бесконечна в обе стороны и разбита на клетки, которые никак не нумеруются и не именуются. В каждой клетке может быть записан один символ или ничего не записано. Содержимое клетки может меняться - в неё можно записать другой символ или стереть находящийся там символ.

Договоримся пустое содержимое клетки называть символом «пусто» и обозначать знаком Л («лямбда»). В связи с этим изображение ленты, показанное на рисунке справа, такое же, как и на рисунке слева. Данное соглашение удобно тем, что операцию стирания символа в некоторой клетке можно рассматривать как запись в эту клетку символа Л, поэтому вместо длинной фразы «записать символ в клетку или стереть находящийся там символ» можно говорить просто «записать символ в клетку».

Автомат - это активная часть МТ. В каждый момент он размещается под одной из клеток ленты и видит её содержимое; это видимая клетка, а находящийся в ней символ - видимый символ; содержимое соседних и других клеток автомат не видит. Кроме того, в каждый момент автомат находится в одном из состояний, которые будем обозначать буквой q с номерами: q1, q2 и т.п. Находясь в некотором состоянии, автомат выполняет какую-то определённую операцию (например, перемещается направо по ленте, заменяя все символы b на a), находясь в другом состоянии - другую операцию.

Пару из видимого символа (S) и текущего состояния автомата (q) будем называть конфигурацией и обозначать <S, q>.

Автомат может выполнять три элементарных действия: 1) записывать в видимую клетку новый символ (менять содержимое других клеток автомат не может); 2) сдвигаться на одну клетку влево или вправо («перепрыгивать» сразу через несколько клеток автомат не может); 3) переходить в новое состояние. Ничего другого делать автомат не умеет, поэтому все более сложные операции так или иначе должны быть сведены к этим трём элементарным действиям.

Такт работы машины Тьюринга МТ работает *тактами,* которые выполняются один за другим. На каждом такте автомат МТ выполняет три следующих действия, причем обязательно в указанном порядке:

1) записывает некоторый символ *S'* в видимую клетку (в частности, может быть записан тот же символ, что и был в ней, тогда содержимое этой клетки не меняется);
2) сдвигается на одну клетку влево (обозначение - L, от *left),* либо на одну клетку вправо (обозначение - R, от *right),* либо остается неподвижным (обозначение - N).
3) переходит в некоторое состояние *q'* (в частности, может остаться в прежнем состоянии).

Формально действия одного такта будем записывать в виде тройки:

S' [lrn], *q'*

где конструкция с квадратными скобками означает возможность записи в этом месте любой из букв L, *R* или N. Например, такт **,L,q8* означает запись символа * в видимую клетку, сдвиг на одну клетку влево и переход в состояние *q8.*

Программа для машины Тьюринга Сама по себе МТ ничего не делает. Для того чтобы заставить её работать, надо написать для неё *программу.* Эта программа записывается в виде следующей таблицы:

	S_1	S_2	…	S_i	…	S_n	Λ
q_1							
…							
q_i				S´, [L, R, N], q´			
…							
q_m							

Слева перечисляются все состояния, в которых может находиться автомат, сверху - все символы (в том числе и Л), которые автомат может видеть на ленте. (Какие именно символы и состояния указывать в таблице - определяет автор программы.) На пересечениях же (в ячейках таблицы) указываются те такты, которые должен выполнить автомат, когда он находится в

соответствующем состоянии и видит на ленте соответствующий символ.

В целом таблица определяет действия МТ при всех возможных конфигурациях и тем самым полностью задаёт поведение МТ. Описать алгоритм в виде МТ - значит предъявить такую таблицу.

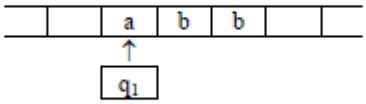
(Замечание. Часто МТ определяют как состоящую из ленты, автомата и программы, поэтому при разных программах получаются разные МТ. Мы же будет считать, в духе современных компьютеров, что МТ одна, но она может выполнять разные программы.)

Правила выполнения программы

До выполнения программы нужно проделать следующие предварительные действия.

Во-первых, надо записать на ленту *входное слово,* к которому будет применена программа. Входное слово - это конечная последовательность символов, записанных в соседних клетках ленты; внутри входного слова пустых клеток быть не должно, а слева и справа от него должны быть только пустые клетки. Пустое входное слово означает, что все клетки ленты пусты.

Во-вторых, надо установить автомат в состояние q_1 (указанное в таблице первым) и разместить его под первым символом входного слова:



Если входное слово пустое, то автомат может смотреть в любую клетку, т.к. все они пусты.

После этих предварительных действий начинается выполнение программы. В таблице отыскивается ячейка на пересечении первой строки (т.к. автомат находится в состоянии q_1) и того столбца, который соответствует первому символу входного слова (это необязательно левый столбец таблицы), и выполняется такт, указанный в этой ячейке. В результате автомат окажется в новой конфигурации. Теперь такие же действия повторяются, но уже для новой конфигурации: в таблице отыскивается ячейка, соответствующая состоянию и символу этой конфигурации, и выполняется такт из этой ячейки. И так далее.

Когда завершается выполнение программы? Введём понятие *такта останова*. Это такт, который ничего не меняет: автомат записывает в видимую клетку тот же символ, что и был в ней раньше, не сдвигается и остается в прежнем состоянии, т.е. это такт *S,N,q* для конфигурации <S, q>. Попав на такт останова, МТ, по определению, останавливается, завершая свою работу.

В целом возможны два исхода работы МТ над входным словом:

1) Первый исход - «хороший»: это когда в какой-то момент МТ останавливается (попадает на такт останова). В таком случае говорят, что МТ *применима* к заданному входному слову. А то слово, которое к этому моменту получено на ленте, считается *выходным словом,* т.е. результатом работы МТ, ответом.

В момент останова должны быть выполнены следующие обязательные условия:

- внутри выходного слова не должно быть пустых клеток (отметим, что во время выполнения программы внутри обрабатываемого слова пустые клетки могут быть, но в конце их уже не должно остаться);

автомат обязан остановиться под одним из символов выходного слова (под каким именно - не играет роли), а если слово пустое - под любой клеткой ленты.

2) Второй исход - «плохой»: это когда МТ зацикливается, никогда не попадая на такт останова (например, автомат на каждом шаге сдвигается вправо и потому не может остановиться, т.к. лента бесконечна). В этом случае говорят, что МТ *неприменима* к заданному входному слову. Ни о каком результате при таком исходе не может идти и речи.

Отметим, что один и тот же алгоритм (программа МТ) может быть применимым к одним входным словам (т. е. останавливаться) и неприменимым к другим (т.е. зацикливаться). Таким образом, применимость/неприменимость зависит не только от самого алгоритма, но и от входного слова.

На каких входных словах алгоритм должен останавливаться? На, так сказать, хороших словах, т.е. на тех, которые относятся к допустимым исходным данным решаемой задачи, для которых задача осмысленна. Но на ленте могут быть записаны любые входные слова, в том числе и те, для которых задача не имеет смысла; на таких словах поведение алгоритма не фиксируется, он может остановиться (при любом результате), а может и зациклиться.

Соглашения для сокращения записи

Договоримся о некоторых соглашениях, сокращающих запись программы для МТ.

1) Если в такте не меняется видимый символ, или автомат не сдвигается, или не

меняется состояние автомата, то в соответствующей позиции такта мы не будем ничего писать.

Например, при конфигурации <*a, q1*> следующие записи тактов эквивалентны:

a,R,q3 ≡ ,R,q3 (но не Λ,R,q3 !!)

b,N,q2 ≡ b, ,q2

a,L,q1 ≡ ,L,

a,N,q1 ≡ , , (это такт останова)

Замечание. Запятые в тактах желательно не опускать, т.к. иначе возможна путаница, если среди символов на ленте могут встретиться буквы *L* и R.

2) Если надо указать, что после выполнения некоторого такта МТ должна остановиться, то в третьей позиции этого такта будем писать знак «!». Например, такт b,L,! означает следующие действия: запись символа *b* в видимую клетку ленты, сдвиг влево и останов.

Формально можно считать, что в программе МТ имеется состояние с названием !, во всех ячейках которого записаны такты останова. При этом, однако, такую строку явно не выписывают, а лишь подразумевают.

3) Если заранее известно, что в процессе выполнения программы не может появиться некоторая конфигурация, тогда, чтобы подчеркнуть это явно, будем в соответствующей ячейке таблицы рисовать крестик. (Формально этот крестик считается тактом останова.)

Эти соглашения необязательны, но они сокращают запись программы и упрощают её восприятие.

Для сокращения формулировки задач введём следующие два соглашения:

- буквой *P* будем обозначать входное слово;
- буквой *A* будем обозначать алфавит входного слова, т.е. набор тех символов, из которых и только которых может состоять *P* (отметим, однако, что в промежуточных и выходном словах могут появляться и другие символы).

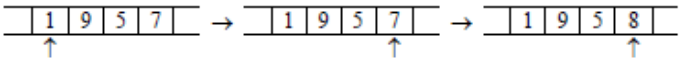
Пример 1 (перемещение автомата, замена символов) A={0,1,2,3,4,5,6,7,8,9}. Пусть *P* - непустое слово; значит, *P* - это последовательность из десятичных цифр, т.е. запись неотрицательного целого числа в десятичной системе. Требуется получить на ленте запись числа, которое на 1 больше числа *P*.

Решение.

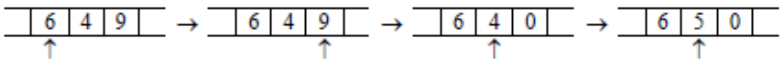
Для решения этой задачи предлагается выполнить следующие действия:

1. Перегнать автомат под последнюю цифру числа.

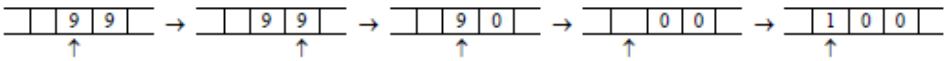
Если это цифра от 0 до 8, то заменить её цифрой на 1 больше и остановиться; например:



3. Если же это цифра 9, тогда заменить её на 0 и сдвинуть автомат к предыдущей цифре, после чего таким же способом увеличить на 1 эту предпоследнюю цифру; например:



4. Особый случай: в *P* только девятки (например, 99). Тогда автомат будет сдвигаться влево, заменяя девятки на нули, и в конце концов окажется под пустой клеткой. В эту пустую клетку надо записать 1 и остановиться (ответом будет 100):



В виде программы для МТ эти действия описываются следующим образом:

	0	1	2	3	4	5	6	7	8	9	Λ
q1	0,R,q1	1,R,q1	2,R,q1	3,R,q1	4,R,q1	5,R,q1	6,R,q1	7,R,q1	8,R,q1	9,R,q1	Λ,L,q2
q2	1,N,!	2, N,!	3, N,!	4, N,!	5, N,!	6, N,!	7, N,!	8, N,!	9, N,!	0,L,q2	1,N,!

Пояснения.

q1 - это состояние, в котором автомат «бежит» под последнюю цифру числа. Для этого он всё время движется вправо, не меняя видимые цифры и оставаясь в том же состоянии. Но здесь есть одна особенность: когда автомат находится под

последней цифрой, то он ещё не знает об этом (ведь он не видит, что записано в соседних клетках) и определит это лишь тогда, когда попадёт на пустую клетку. Поэтому, дойдя до первой пустой клетки, автомат возвращается назад под последнюю цифру и переходит в состояние *q2* (вправо двигаться уже не надо).

q2 - это состояние, в котором автомат прибавляет 1 к той цифре, которую видит в данный момент. Сначала это последняя цифра числа; если она - в диапазоне от 0 до 8, то автомат заменяет её цифрой, которая на 1 больше, и останавливается. Но если это цифра 9, то автомат заменяет её на 0 и сдвигается влево, оставаясь в состоянии *q2*. Тем самым, он будет теперь прибавлять 1 к предыдущей цифре. Если и эта цифра равна 9, то автомат заменяет её на 0 и сдвигается влево, оставаясь по- прежнему в состоянии *q2,* т.к. должен выполнить то же самое действие - увеличить на 1 видимую цифру. Если же автомат сдвинулся влево, а в видимой клетке нет цифры (а есть «пусто»), то он записывает сюда 1 и останавливается.

Отметим, что для пустого входного слова наша задача не определена, поэтому на этом слове МТ может вести себя как угодно. В нашей программе, например, при пустом входном слове МТ останавливается и выдает ответ 1.

Выше мы привели запись программы в полном, несокращённом виде. Теперь же приведём запись программы в сокращённом, более наглядном виде, при этом справа дадим краткое пояснение действий, которые реализуются в соответствующих состояниях автомата:

	0	1	2	3	4	5	6	7	8	9	Λ	
q1	,R,	,R,	,R,	,R,	,R,	,R,	,R,	,R,	,R,	,R,	,L,q2	под последнюю цифру
q2	1, ,!	2, ,!	3, ,!	4, ,!	5, ,!	6, ,!	7, ,!	8, ,!	9, ,!	0,L,	1, ,!	видимая цифра + 1

Именно так мы и будем в дальнейшем записывать программы.

Пример 2 (анализ символов)

$A=\{a,b,c\}$. Перенести первый символ непустого слова *P* в его конец.

Например:

	a	b	c	b			→			b	c	b	a				

Решение.

Для решения этой задачи предлагается выполнить следующие действия:

1. Запомнить первый символ слова *P,* а затем стереть этот символ.
2. Перегнать автомат вправо под первую пустую клетку за *P* и записать в неё запомненный символ.

Как «бегать» вправо, мы уже знаем из предыдущего примера. А вот как запомнить первый символ? Ведь в МТ нет другого запоминающего устройства, кроме ленты, а запоминать символ в какой-то клетке на ленте бессмысленно: как только автомат сдвинется влево или вправо от этой клетки, он тут же забудет данный символ. Что делать?

Выход здесь таков - надо использовать разные состояния автомата. Если первый символ - это a, то надо перейти в состояние *q2,* в котором автомат бежит вправо и записывает в конце *a.* Если же первым был символ *b,* тогда надо перейти в состояние *q3*, где делается всё то же самое, только в конце записывается символ *b.* Если же первым был символ *c*, тогда переходим в состояние *q4*, в котором автомат дописывает за входным словом символ *c.* Следовательно, то, каким был первый символ, мы фиксируем переводом автомата в разные состояния. Это типичный приём при программировании на МТ.

С учётом сказанного программа будет такой:

	a	b	c	Λ	
q1	Λ,R,q2	Λ,R,q3	Λ,R,q4	,R,	анализ 1-го символа, удаление его, разветвление
q2	,R,	,R,	,R,	a, ,!	запись *a* справа
q3	,R,	,R,	,R,	b, ,!	запись *b* справа
q4	,R,	,R,	,R,	c, ,!	запись *c* справа

Задание:

В задачах рассматриваются только целые неотрицательные числа, если не сказано иное.

Под «единичной» системой счисления понимается запись неотрицательного целого числа с помощью палочек - должно быть выписано столько палочек, какова величина числа; например: 2^ | | , 5 ^ | | | | | , 0 ^ <пустое слово>.

1. *A={a,b,c}.* Приписать слева к слову *P* символ *b (P ^ bP).*
2. A={a,b,c}. Приписать справа к слову *P* символы *bc (P ^ Pbc).*
3. A={a,b,c}. Заменить на *a* каждый второй символ в слове P.

4. A={a, b,c}. Оставить в слове *P* только первый символ (пустое слово не менять).

5. A={a, b,c}. Оставить в слове *P* только последний символ (пустое слово не менять).

6. A={a,b,c}. Определить, является ли *P* словом *aB*. Ответ (выходное слово): слово *aB*, если является, или пустое слово иначе.

7. A={a,b,c}. Определить, входит ли в слово *P* символ a. Ответ: слово из одного символа *a* (да, входит) или пустое слово (нет).

8. A={a,b,c}. Если в слово *P* не входит символ a, то заменить в *P* все символы *B* на *c,* иначе в качестве ответа выдать слово из одного символа *a.*

9. A={a,b,0,1}. Определить, является ли слово *P* идентификатором (непустым словом, начинающимся с буквы). Ответ: слово *a* (да) или пустое слово (нет).

10. A={a,b,0,1}. Определить, является ли слово *P* записью числа в двоичной системе счисления (непустым словом, состоящем только из цифр 0 и 1). Ответ: слово 1 (да) или слово 0.

Содержание отчета

– Название работы;
– Цель работы;
– Формулировку задания;
– Решенные задания, с пояснением решения в виде таблиц, рассуждений, рисунков.

Контрольные вопросы

1. Перечислите элементы в устройстве машины Тьюринга и основные команды.
2. Какие свойства алгоритма позволяет продемонстрировать машинаТьюринга?

Список литературы:

Основная литература

1. Окулов С. М. Программирование в алгоритмах. / М. Окулов. – М.: БИНОМ. Лаборатория знаний, 2002. – 341 с.
2. Иванов Б. Н. Дискретная математика. Алгоритмы и программы. – М.: Лаборатория Базовых Знаний, 2002.

Дополнительная литература

1. Семакин И. Г. Основы программирования: учебник для сред. проф. образования / И. Г. Семакин, А. П. Шестаков. – М.: Издательский центр «Академия», 2007.
2. Акимов О.Е. Дискретная математика: логика, группы, графы. – М.: Лаборатория Базовых Знаний, 2001.

Практическая работа № 3

Тема: Реализация алгоритмов с помощью нормальных алгорифмов Маркова.

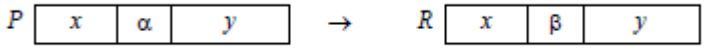
Цель: закрепление умений по реализации с помощью нормальных алгорифмов Маркова.

Пояснения к работе:

Интересной особенностью нормальных алгоритмов Маркова (НАМ) является то, что в них используется лишь одно элементарное действие - так называемая подстановка, которая определяется следующим образом.

Формулой подстановки называется запись вида a^P (читается «a заменить на P»), где a и P - любые слова (возможно, и пустые). При этом a называется левой частью формулы, а P - правой частью.

Сама *подстановка* (как действие) задается формулой подстановки и применяется к некоторому слову *P*. Суть операции сводится к тому, что в слове *P* отыскивается часть, совпадающая с левой частью этой формулы (т.е. с a), и она заменяется на правую часть формулы (т.е. на P). При этом остальные части слова *P* (слева и справа от a) не меняются. Получившееся слово *R* называют *результатом подстановки.* Условно это можно изобразить так:



Необходимые уточнения:

1. Если левая часть формулы подстановки входит в слово P, то говорят, что эта формула *применима к P*. Но если a не входит в P, то формула считается *неприменимой к P*, и подстановка не выполняется.

2. Если левая часть а входит в *P* несколько раз, то на правую часть Р, по определению, заменяется только первое вхождение а в *P*:

P	*x*	α	*y*	α	*z*	→	*R*	*x*	β	*y*	α	*z*

3. Если правая часть формулы подстановки – пустое слово, то подстановка α→ сводится к вычеркиванию части α из *P* (отметим попутно, что в формулах подстановки не принято как-либо обозначать пустое слово):

P	*x*	α	*y*	→	*R*	*x*	*y*

4. Если в левой части формулы подстановки указано пустое слово, то подстановка →β сводится, по определению, к приписыванию β слева к слову *P*:

P	*x*	→	*R*	β	*x*

Из этого правила вытекает очень важный факт: формула с пустой левой частью применима к любому слову. Отметим также, что формула с пустыми левой и правой частями не меняет слово.

Определение НАМ

Нормальным алгоритмом Маркова (НАМ) называется непустой конечный упорядоченный набор формул подстановки:

$$\begin{cases} \alpha_1 \to \beta_1 \\ \alpha_2 \to \beta_2 \\ \quad \dots \\ \alpha_k \to \beta_k \end{cases} \qquad (k \geq 1)$$

В этих формулах могут использоваться два вида стрелок: обычная стрелка (а) и стрелка «с хвостиком» (а). Формула с обычной стрелкой называется *обычной формулой,* а формула со стрелкой «с хвостиком» - *заключительной формулой.* Разница между ними объясняется чуть ниже.

Записать алгоритм в виде НАМ - значит предъявить такой набор формул.

Правила выполнения НАМ

Прежде всего, задается некоторое *входное слово P.* Где именно оно записано - не важно, в НАМ этот вопрос не оговаривается.

Работа НАМ сводится к выполнению последовательности шагов. На каждом шаге входящие в НАМ формулы подстановки просматриваются сверху вниз и выбирается первая из формул, применимых к входному слову *P*, т. е. самая верхняя из тех, левая часть которых входит в *P*. Далее выполняется подстановка согласно найденной формуле. Получается новое слово *P′*

На следующем шаге это слово *P′* берется за исходное и к нему применяется та же самая процедура, т.е. формулы снова просматриваются сверху вниз начиная с самой верхней и ищется первая формула, применимая к слову *P′,* после чего выполняется соответствующая подстановка и получается новое слово *P* " И так далее:

$$P \to P' \to P'' \to \dots$$

Следует обратить особое внимание на тот факт, что на каждом шаге формулы в НАМ всегда просматриваются начиная с самой первой.

Необходимые уточнения:

1. Если на очередном шаге была применена обычная формула (аа-Р), то работа НАМ продолжается.

2. Если же на очередном шаге была применена заключительная формула (а а Р), то после её применения работа НАМ прекращается. То слово, которое получилось в этот момент, и есть *выходное слово,* т.е. результат применения НАМ к входному слову.

Как видно, разница между обычной и заключительной формулами подстановки проявляется лишь в том, что после применения обычной формулы работа НАМ продолжается, а после заключительной формулы - прекращается.

3. Если на очередном шаге к текущему слову неприменима ни одна формула, то и в этом случае работа НАМ прекращается, а выходным словом считается текущее слово.

Таким образом, НАМ останавливается по двум причинам: либо была применена заключительная формула, либо ни одна из формул не подошла. То и другое считается «хорошим» окончанием работы НАМ. В обоих случаях говорят, что НАМ *применим* к входному слову.

Однако может случиться и так, что НАМ никогда не остановится; это происходит, когда на

каждом шаге есть применимая формула и эта формула обычная. Тогда говорят, что НАМ *неприменим* к входному слову. В этом случае ни о каком результате нет и речи.

1.1 Примеры на составление НАМ

Рассмотрим примеры, в которых демонстрируются типичные приёмы составления НАМ.

Как и в случае машины Тьюринга, для сокращения формулировки задач будем использовать следующие соглашения:

- буквой *P* будем обозначать входное слово;
- буквой *A* будем обозначать алфавит входного слова, т.е. набор тех символов, которые и только которые могут входить во входное слово *P* (но в процессе выполнения НАМ в обрабатываемых словах могут появляться и другие символы).

Кроме того, в примерах будем справа от формул подстановки указывать их номера. Эти номера не входят в формулы, а нужны для ссылок на формулы при показе пошагового выполнения НАМ.

Пример 1 (вставка и удаление символов)

A={a,b,c,d}. В слове *P* требуется заменить первое вхождение подслова *bb* на *ddd* и удалить все вхождения символа *c*.

Например: abbcabbca — adddabba

Решение.

Прежде всего отметим, что в НАМ, в отличие от машины Тьюринга, легко реализуются вставки и удаления символов. Вставка новых символов в слово - это замена некоторого подслова на подслово с большим числом символов; например, с помощью формулы *bb—ddd* два символа будут заменены на три символа. При этом не надо заботиться о том, чтобы предварительно освободить место для дополнительных символов, в НАМ слово раздвигается автоматически. Удаление же символов - это замена некоторого подслова на подслово с меньшим числом символов; например, удаление символа *c* реализуется формулой *c*—— (с пустой правой частью). При этом никаких пустых позиций внутри слова не появляется, сжатие слова в НАМ происходит автоматически.

С учётом сказанного нашу задачу должен, казалось бы, решать такой НАМ:

$$\begin{cases} bb \to ddd & (1) \\ c \to & (2) \end{cases}$$

Однако это не так. Проверим этот НАМ на входном слове *abbcabbca* (над стрелками указаны номера применённых формул, а в словах слева от стрелок подчёркнуты для наглядности те части, к которым были применены эти формулы):

$$\underline{abb}cabbca \xrightarrow{1} adddca\underline{bb}ca \xrightarrow{1} adddc\underline{a}dddca \xrightarrow{2} adddab\underline{bca} \to \ldots$$

Как видно, заменив первое вхождение *bb* на *ddd,* этот НАМ не перешёл сразу к удалению символов *c,* а стал заменять и другие вхождения *bb.* Почему? Напомним, что на каждом шаге работы НАМ формулы подстановки всегда просматриваются сверху вниз начиная с первой из них. Поэтому, пока применима первая формула, она и будет применяться, блокируя доступ к остальным формулам. Этот означает, что в НАМ важен порядок перечисления формул подстановки.

Учтём это и переставим наши две формулы:

$$\begin{cases} c \to & (1) \\ bb \to ddd & (2) \end{cases}$$

Проверим этот новый алгоритм на том же входном слове:

$$abb\underline{c}abbca \xrightarrow{1} abbabb\underline{ca} \xrightarrow{1} a\underline{bb}abba \xrightarrow{2} adddab\underline{ba} \xrightarrow{2} adddaddda$$

Итак, НАМ сначала удалил все символы *c* и только затем заменил первое вхождение *bb* на *ddd.* Однако НАМ на этом не остановился и стал заменять остальные вхождения *bb.* Почему? Дело в том, что, пока применима хотя бы одна формула, НАМ продолжает свою работу. Но нам этого не надо, поэтому мы должны принудительно остановить НАМ после того, как он заменил первое вхождение bb. Вот для этого и нужны заключительные формулы подстановки, после применения которых НАМ останавливается. Следовательно, в нашем алгоритме обычную формулу *bb-ddd* надо заменить на заключительную формулу *bb -ddd:*

$$\begin{cases} c \to & (1) \\ bb \mapsto ddd & (2) \end{cases}$$

Вот теперь наш алгоритм будет работать правильно:

$$\text{abbcabbca} \xrightarrow{1} \text{abbabbca} \xrightarrow{1} \text{abbabba} \overset{2}{\mapsto} \text{adddabba}$$

Слово, которое получилось после применения заключительной формулы (2), является выходным словом, т.е. результатом применения НАМ к заданному входному слову.

Проверим наш НАМ ещё и на входном слове, в которое не входит *bb*:

$$\text{dcacb} \xrightarrow{1} \text{dacb} \xrightarrow{1} \text{dab}$$

К последнему слову (*dab*) неприменима ни одна формула, поэтому, согласно определению НАМ, алгоритм останавливается и это слово объявляется выходным.

Пример 2 (перестановка символов)

A={*a*,*b*}. Преобразовать слово *P* так, чтобы в его начале оказались все символы *a*, а в конце – все символы *b*.

Например: babba → aabbb

Решение.

Казалось бы, для решения этой задачи нужен сложный НАМ. Однако это не так, задача решается с помощью НАМ, содержащего всего одну формулу:

$$\{ ba \to ab$$

Пока в слове *P* справа хотя бы от одного символа *b* есть символ *a*, эта формула будет переносить *a* налево от этого *b*. Формула перестает работать, когда справа от *b* нет ни одного *a*, это и означает, что все *a* оказались слева от *b*. Например:

babba → abbba → abbab → ababb → aabbb

Алгоритм остановился на последнем слове, т.к. к нему уже неприменима наша формула.

Этот и предыдущий примеры показывают, что в НАМ, в отличие от машины Тьюринга, легко реализуются перестановки, вставки и удаления символов. Однако в НАМ возникает другая проблема: как зафиксировать символ (подслово), который должен быть обработан? Рассмотрим эту проблему на следующем примере.

Задание:

Замечания:

1) В задачах рассматриваются только целые неотрицательные числа, если не сказано иное.

2) Под «единичной» системой счисления понимается запись неотрицательного целого числа с помощью палочек - должно быть выписано столько палочек, какова величина числа; например: 2а | | , 5 а | | | | | , 0 а <пустое слово>.

1. *A={f,h,p}.* В слове *P* заменить все пары*ph* на *f*
2. *A={f,h,p}.* В слове *P* заменить на *f* только первую пару *ph,* если такая есть.
3. A={a,b,c}. Приписать слово *bac* слева к слову P.
4. A={a,b,c}. Заменить слово *P* на пустое слово, т.е. удалить из *P* все символы.
5. A={a,b,c}. Заменить любое входное слово на слово a.
6. Выписать НАМ, не меняющий входное слово (при любом алфавите A).
7. A={ | }. Считая слово *P* записью числа в единичной системе счисления, получить остаток от деления этого числа на 2, т.е. получить слово из одной палочки, если число нечётно, или пустое слово, если число чётно.
8. A={ | }. Считая слово *P* записью положительного числа в единичной системе счисления, уменьшить это число на 1.

9. A={ | }. Считая слово *P* записью числа в единичной системе счисления, увеличить это число на 2.

10. A={0,1,2}. Считая слово *P* записью числа в троичной системе счисления, получить остаток от деления этого числа на 2, т.е. получить слово 1, если число нечётно, или слово 0, если число чётно. *(Замечание:* в чётном троичном числе должно быть чётное количество цифр 1.)

Содержание отчета

– Название работы;
– Цель работы;
– Формулировку задания;
– Решенные задания, с пояснением решения в виде таблиц, рассуждений, рисунков.

Контрольные вопросы

1. В чем состоит принцип составления алгоритмов для НАМ?
2. Какие свойства алгоритма демонстрируют НАМ?

Список литературы:

Основная литература

1. Окулов С. М. Программирование в алгоритмах. / М. Окулов. – М.: БИНОМ. Лаборатория знаний, 2002. – 341 с.
2. Иванов Б. Н. Дискретная математика. Алгоритмы и программы. – М.: Лаборатория Базовых Знаний, 2002.

Дополнительная литература

1. Семакин И. Г. Основы программирования: учебник для сред. проф. образования / И. Г. Семакин, А. П. Шестаков. – М.: Издательский центр «Академия», 2007.
2. Акимов О.Е. Дискретная математика: логика, группы, графы. – М.: Лаборатория Базовых Знаний, 2001.

Практическая работа № 4

Тема: Реализация алгоритмов с помощью нормальных алгорифмов Маркова.

Цель: закрепление умений по реализации с помощью нормальных алгорифмов Маркова.

Пояснения к работе:

В НАМ, в отличие от машины Тьюринга, легко реализуются перестановки, вставки и удаления символов. Однако в НАМ возникает другая проблема: как зафиксировать символ подслово), который должен быть обработан?

Пример 3 (использование спецзнака)

A={a,b}. Удалить из непустого слова *P* его первый символ. Пустое слово не менять.

Решение.

Ясно, что удалив первый символ слова, надо тут же остановиться. Поэтому, казалось бы, задачу решает следующий НАМ:

$$\begin{cases} a \mapsto & (1) \\ b \mapsto & (2) \end{cases}$$

Однако это неправильный алгоритм, в чём можно убедиться, применив его к слову *bbaba*:

$$\text{bb}\underline{\text{a}}\text{ba} \overset{1}{\mapsto} \text{bbba}$$

Как видно, этот НАМ удалил не первый символ слова, а первое вхождение символа *a,* а это разные вещи. Данный алгоритм будет правильно работать, только если входное слово начинается с символа *a.* Ясно, что перестановка формул в этом НАМ не поможет, т.к. тогда он будет, напротив, неправильно работать на словах, начинающихся с *a.*

Что делать? Надо как-то зафиксировать, пометить первый символ слова, например, поставив перед ним какой-либо знак, скажем *, отличный от символов алфавита слова. После этого уже можно с помощью формул вида *E, а заменить этот знак и первый символ E, слова на пусто и остановиться: bbaba - *bbaba - baba

А как поставить * перед первым символом? Это реализуется формулой a* с пустой левой частью, которая, по определению, приписывает свою правую часть слева к слову.

Итого, получаем следующий НАМ:

$$\begin{cases} \to * & (1) \\ *a \mapsto & (2) \\ *b \mapsto & (3) \end{cases}$$

Проверим его на том же входном слове:

$$\text{bbaba} \xrightarrow{1} \text{*bbaba} \xrightarrow{1} \text{**bbaba} \xrightarrow{1} \text{***bbaba} \xrightarrow{1} \ldots$$

Как видно, этот алгоритм постоянно приписывает слева звёздочки. Почему? Напомним, что формула постановки с пустой левой частью применима всегда, поэтому наша формула (1) будет работать бесконечно, блокируя доступ к остальным формулам. Отсюда вытекает очень важное правило: если в НАМ есть формула с пустой левой частью (aP), то её место - только в самом конце НАМ. Учтём это правило и перепишем наш НАМ:

$$\begin{cases} *a \mapsto & (1) \\ *b \mapsto & (2) \\ \to * & (3) \end{cases}$$

Проверим данный алгоритм:

$$\text{bbaba} \xrightarrow{3} \underline{\text{*b}}\text{baba} \xmapsto{2} \text{baba}$$

Казалось бы, всё в порядке. Однако это не так: наш алгоритм зациклится на пустом входном слове, т.к. постоянно будет применяться формула (3), а согласно условию задачи на таком слове НАМ должен остановиться. В чём причина этой ошибки? Дело в том, что мы ввели знак * для того, чтобы пометить первый символ слова, а затем уничтожить * и этот символ. Но в пустом слове нет ни одного символа, поэтому формулы (1) и (2) ни разу не сработают и постоянно будет выполняться формула (3). Следовательно, чтобы учесть случай пустого входного слова, надо после формул (1) и (2) записать ещё одну формулу, которая уничтожает «одинокую» звёздочку и останавливает алгоритм:

$$\begin{cases} *a \mapsto & (1) \\ *b \mapsto & (2) \\ * \mapsto & (3) \\ \to * & (4) \end{cases}$$

обобщим тот приём со звёздочкой, который мы использовали в примере 3.

Пусть в обрабатываемое слово *P* входит несколько раз подслово a :

P						

и нам надо заменить одно из вхождений a на подслово p. Такая замена делается с помощью формулы aap. Однако, если мы применим эту формулу к слову P, то будет заменено первое вхождение a. А что делать, если надо заменить какое- другое вхождение a , скажем второе или последнее? Так вот, чтобы на P заменялось не первое вхождение a , а какое-то другое, это другое вхождение надо как-то выделить, пометить, для чего следует рядом с ним (слева или справа) поставить некоторый символ, скажем *, отличный от всех других символов, входящих в *P*:

P						

Такой символ будем в дальнейшем называть *спецзнаком.* Его роль - выделить нужное вхождение a среди других, сделать его уникальным. Поскольку только около этого вхождения есть спецзнак, то надо использовать формулу *aap, чтобы заменить на в именно это вхождение a , а не какое-то другое.

Этот приём со спецзнаком следует запомнить, т. к. в НАМ он используется очень часто.

Правда, остаётся ещё одна проблема: как спецзнак разместить рядом с нужным вхождением a ? Следующие примеры показывают, как это делается.

Пример 4 (фиксация спецзнаком заменяемого символа)

A={0,1,2,3}. Пусть *P* - непустое слово. Трактуя его как запись неотрицательного целого числа в четверичной системе счисления, требуется получить запись этого же числа, но в двоичной системе. **Например:** 0123 → 00011011

Решение.

Как известно, для перевода числа из четверичной системы в двоичную надо каждую четверичную цифру заменить на пару соответствующих ей двоичных

ных цифр: 0→00, 1→01, 2→10, 3→11. Такая замена, казалось бы, реализуется следующим НАМ:

$$\begin{cases} 0 \to 00 & (1) \\ 1 \to 01 & (2) \\ 2 \to 10 & (3) \\ 3 \to 11 & (4) \end{cases}$$

Но этот алгоритм неправильный, в чём можно убедиться на входном слове *0123*:

$$\underline{0}123 \xrightarrow{1} \underline{0}0123 \xrightarrow{1} \underline{0}00123 \xrightarrow{1} \ldots$$

Ошибка здесь в том, что после замены четверичной цифры на пару двоичных цифр уже никак нельзя отличить двоичные цифры от четверичных, поэтому наш НАМ начинает заменять и двоичные цифры. Значит, надо как-то отделить ту часть числа, в которой уже была произведена замена, от той части, где замены ещё не было. Для этого предлагается пометить слева спецзнаком * ту четверичную цифру, которая сейчас должна быть заменена на пару соответствующих двоичных цифр, а после того как такая замена будет выполнена, спецзнак нужно поместить перед следующей четверичной цифрой:

$$0123 \to *0123 \to 00*123 \to 0001*23 \to 000110*3 \to 00011011*$$

Как видно, слева от спецзнака всегда находится та часть числа, которая уже переведена в двоичный вид, а справа - часть, которую ещё предстоит заменить. Поэтому никакой путаницы между четверичными и двоичными цифрами уже не будет.

Итак, спецзнак * сначала должен быть размещён слева от первой цифры четверичного числа, а затем он должен «перепрыгивать» через очередную четверичную цифру, оставляя слева от себя соответствующие ей двоичные цифры. В конце же, когда справа от * уже не окажется никакой цифры, спецзнак надо уничтожить и остановиться. Как приписать * слева к входному слову и как уничтожить спецзнак с остановом, мы уже знаем по предыдущему примеру, а вот «перепрыгивание» звёздочки реализуется с помощью формул вида *aaPy*, где a - четверичная цифра, а Py - соответствующая ей пара двоичных цифр.

Итого, получаем следующий алгоритм перевода чисел из четверичной системы в двоичную:

$$\begin{cases} *0 \to 00* & (1) \\ *1 \to 01* & (2) \\ *2 \to 10* & (3) \\ *3 \to 11* & (4) \\ * \mapsto & (5) \\ \quad \to * & (6) \end{cases}$$

Проверим этот НАМ на входном слове 0123:

$$0123 \xrightarrow{6} \underline{*}0123 \xrightarrow{1} 00\underline{*}123 \xrightarrow{2} 0001\underline{*}23 \xrightarrow{3} 000110\underline{*}3 \xrightarrow{4} 00011011\underline{*} \xrightarrow{5} 00011011$$

Задание:

1. A={a,b,c}. Определить, входит ли символ *a* в слово P. Ответ (выходное слово): слово a, если входит, или пустое слово, если не входит.

2. A={a,b}. Если в слово *P* входит больше символов a, чем символов b, то в качестве ответа выдать слово из одного символа *a*, если в *P* равное количество *a* и *b*, то в качестве ответа выдать пустое слово, а иначе выдать ответ *b*.
3. A={0,1,2,3}. Преобразовать слово *P* так, чтобы сначала шли все чётные цифры (0 и 2), а затем - все нечётные.
4. A={a,b,c}. Преобразовать слово *P* так, чтобы сначала шли все символы a, затем - все символы *b* и в конце - все символы c.
5. A={a,b,c}. Определить, из скольких различных символов составлено слово P; ответ получить в единичной системе счисления (например: *acaac* a | |).
6. A={a,b,c}. В непустом слове *P* удвоить первый символ, т.е. приписать этот символ слева к *P*.
7. A={a,b,c}. За первым символом непустого слова *P* вставить символ c.
8. A={a,b,c}. Из слова *P* удалить второй символ, если такой есть.
9. A={a,b,c}. Если в слове *P* не менее двух символов, то переставить два первых символа.
10. A={0,1,2}. Считая непустое слово *P* записью троичного числа, удалить из этой записи все незначащие нули.

11. A={a,b,c}. Приписать слово *abc* справа к слову P.
12. A={a,b,c}. Удалить из непустого слова *P* его последний символ.
13. A={0,1}. Считая непустое слово *P* записью числа в двоичной системе, получить двоичное число, равное учетверённому числу *P* (например: 101 а 10100).
14. A={0,1}. Считая непустое слово *P* записью числа в двоичной системе, получить двоичное число, равное неполному частному от деления числа *P* на 2 (например: 1011 а 101).
15. A={a,b}. В слове *P* все символы *a* заменить на b, а все (прежние) символы *b* - на *a*.
16. A={a,b,c}. Удвоить каждый символ в слове *P* (например: *bacb* а *bbaaccbb).*
17. A={a,b}. Приписать справа к слову *P* столько палочек, сколько всего символов входит в *P* (например: *babb* а *babb* | | | |) .
18. A={a,b}. Пусть слово *P* имеет чётную длину (0, 2, 4, ...). Удалить правую половину этого слова. *(Рекомендация:* использовать решение предыдущей задачи.)
19. A={a,b}. Пусть длина слова *P* кратна 3. Удалить правую треть этого слова.
20. A={a,b}. Приписать справа к слову *P* столько палочек, со скольких подряд идущих символов *a* начинается это слово (например: *aababa* а *aababa*| |).

Содержание отчета

– Название работы;
– Цель работы;
– Формулировку задания;
– Решенные задания, с пояснением решения в виде таблиц, рассуждений, рисунков.

Контрольные вопросы

1. Как в НАМ реализуются перестановки символов?
2. Как в НАМ реализуются удаление символов?
3. Как в НАМ реализуются вставка символов?

Список литературы:

Основная литература

1. Окулов С. М. Программирование в алгоритмах. / М. Окулов. – М.: БИНОМ. Лаборатория знаний, 2002. – 341 с.
2. Иванов Б. Н. Дискретная математика. Алгоритмы и программы. – М.: Лаборатория Базовых Знаний, 2002.

Дополнительная литература

1. Семакин И. Г. Основы программирования: учебник для сред. проф. образования / И. Г. Семакин, А. П. Шестаков. – М.: Издательский центр «Академия», 2007.
2. Акимов О.Е. Дискретная математика: логика, группы, графы. – М.: Лаборатория Базовых Знаний, 2001.

Практическая работа № 5

Тема: Реализация метода перебора с возвратом.

Цель: закрепление умений по программированию алгоритмов.

Пояснения к работе:

Во многих практических задачах из различных предметных областей требуется найти общее количество вариантов решения, число элементов в полном наборе решений. Иногда, исходя из постановки задачи, достаточно найти один из вариантов, соответствующих условию задачи. В некоторых задачах изучается вопрос о существовании решения как такового.

Ответы на поставленные вопросы, как правило, требуют проведения исчерпывающего поиска в некотором множестве всех возможных вариантов, среди которых находятся решения конкретной задачи. Существуют два общих метода организации исчерпывающего поиска: перебор с возвратом (backtracking) и его естественное логическое дополнение – метод решета.

Решение задачи методом перебора с возвратом строится конструктивно последовательным расширением частичного решения. Если на конкретном шаге такое расширение провести не удается, то происходит возврат к более короткому частичному решению, и попытки его расширить продолжаются. Для ускорения перебора с возвратом вычисления всегда стараются организовать так, чтобы была возможность отказаться как можно раньше от как можно большего

числа заведомо неподходящих вариантов. Незначительные модификации метода перебора с возвратом, связанные с представлением данных или особенностями реализации, имеют и иные названия: метод ветвей и границ (branch and bound), поиск в глубину (depth first search), метод проб и ошибок и т. д. Перебор с возвратом практически одновременно и независимо был изобретен многими исследователями еще до его формального описания. Он находит применение при решении различных комбинаторных задач в области искусственного интеллекта.

При использовании метода решета вместо конструктивного построения решений задачи из множества возможных вариантов исключаются все элементы, не являющиеся решениями. Методы решета нашли широкое применение в теоретико-числовых задачах.

Метод перебора с возвратом и метод решета, строго говоря, не являются ни методами, ни алгоритмами решения задач. Их следует воспринимать как на некоторые общие схемы, которые применяются для решения той или иной задачи. Реализация этих схем в виде конкретных алгоритмов часто требует значительных дополнительных усилий в представлении данных и описании зависимостей между ними.

Соединение метода перебора с возвратом и рекурсии определяет специфический способ реализации рекурсивных вычислений и называется возвратной рекурсией. Это соединение двух эффективных методов реализации переборных алгоритмов.

При использовании возвратной рекурсии отпадает необходимость непосредственно организовывать возвраты и отслеживать правильность их осуществления. Они, как правило, становятся встроенной частью механизма выполнения рекурсивных вызовов.

Вычислительная схема перебора с возвратом

Опишем общую постановку класса задач, к которым заведомо применим алгоритм перебора с возвратом.

Пусть $M_0,\ M_1, \dots M_{n-1}$ – n конечных линейно упорядоченных множеств и G – совокупность ограничений (условий), ставящих в соответствие векторам вида $v = (v_0, v_1, \dots, v_k)^{\mathrm{T}}\ (v_j \in M_j;\ j = 0,1,\dots,k;\ k \le n-1)$, булево значение $G(v) \in \{истина, ложь\}$. Векторы $v = (v_0, v_1, \dots, v_k)^T$, для которых $G(v) = истина$, назовем частичными решениями. Пусть, далее, существует конкретное правило P, в соответствии с которым некоторые из частичных решений могут объявляться полными решениями. Тогда возможна постановка следующих поисковых задач.

Найти все полные решения или установить отсутствие таковых.

Найти хотя бы одно полное решение или установить его отсутствие.

Общий метод решения приведенных задач состоит в последовательном покомпонентном наращивании вектора v слева направо, начиная с v_0, и последующих проверках его ограничениями G и правилом P.

В общем случае этот метод приводит к алгоритмам с экспоненциальной временной сложностью, а применяется он в основном к классу так называемых Np-полных задач (задача коммивояжера, задача о рюкзаке и т. д.). Задачи этого класса эквивалентны друг другу в том смысле, что все они разрешимы недетерминированными алгоритмами полиномиальной сложности. Для них известно, что либо все они разрешимы, либо ни одна из них не разрешима детерминированными алгоритмами полиномиальной сложности. Иными словами, если хотя бы для одной из этих задач не существует детерминированного алгоритма, имеющего в худшем случае полиномиальную трудоемкость, то такие алгоритмы не должны существовать и для остальных задач этого класса. Наоборот, если хотя бы для одной из этих задач удалось найти детерминированный алгоритм, имеющий в худшем случае полиномиальную трудоемкость, то подобные алгоритмы существовали бы и для остальных задач этого класса и, более того, их можно было бы построить.

Опишем схему выполнения недетерминированного алгоритма. Пусть алгоритм выполняется до тех пор, пока не доходит до места, с которого должен быть сделан выбор из нескольких альтернатив. Детерминированный алгоритм однозначно осуществит выбор конкретной альтернативы и продолжит работать в соответствии с эти выбором. Недетерминированный алгоритм исследует все возможности одновременно, как бы копируя себя для реализации вычислений по всем альтернативам одновременно. Далее все копии работают независимо друг от друга и по мере необходимости продолжают создавать новые копии. Копия, сделавшая неправильный или безрезультатный выбор прекращает свою работу. Копия, нашедшая решение задачи, объявляет об этом, давая тем самым сигнал другим копиям о прекращении

вычислений. Недетерминированные алгоритмы, являясь весьма полезной и продуктивной абстракцией, рекурсивны по сути, ибо при реализации оператора выбора фактически обращаются сами к себе.

Возможно, что многие задачи, решаемые методом перебора с возвратом, могут быть решены более эффективно другими способами. Однако ценность метода перебора с возвратом в соединении с рекурсией неоспорима. Во-первых, программы решения многих задач строятся по единой схеме, а во-вторых, они компактны и тем самым просты для понимания и усвоения соответствующих идей.

Использование перебора с возвратом при решении задач

Пример 1. Задача о расстановке ферзей на шахматной доске.

Составьте рекурсивную функцию, находящую возможную расстановку n ферзей на шахматной доске размером n×n так, чтобы они не били друг друга (n – натуральное число).

В соответствии с общей схемой алгоритма перебора с возвратом предложенную задачу можно решать по алгоритму «Все расстановки», но завершить выполнение алгоритма при нахождении первой требуемой расстановки или при получении вывода о невозможности получить нужную комбинацию. Согласно алгоритму ферзи расставляются последовательно на вертикалях с номерами от нуля и далее. В процессе выполнения предписания возможны снятия ферзей с доски (возвраты).

Алгоритм «Все расстановки»

Шаг 1. Полагаем D = $\varnothing$, j = 0 (D – множество решений, j – текущий столбец для очередного ферзя).

Шаг 2. Пытаемся в столбце j продвинуть вниз по вертикали или новый (если столбец j пустой), или уже имеющийся там ферзь на ближайшую допустимую строку. Если это сделать не удалось, то переходим к шагу 4.

Шаг 3. Увеличиваем j на 1, то есть переходим к следующему столбцу. Если $j < n-1$, то переходим к шагу 2. В противном случае $j = n-1$, то есть все вертикали уже заняты. Найденное частичное решение запоминаем в множестве D и переходим к шагу 2.

Шаг 4. Уменьшаем j на 1, то есть снимаем ферзь со столбца j и переходим к предыдущему столбцу. Если $j \geq 0$, то выполняем шаг 2. Иначе вычисления прекращаем. Решения задачи находятся в множестве D, которое, может быть и пустым.

Решение задачи для небольших размеров доски можно найти «вручную», но для разработки алгоритма необходимо провести моделирование условия и остановиться на каком-либо представлении данных.

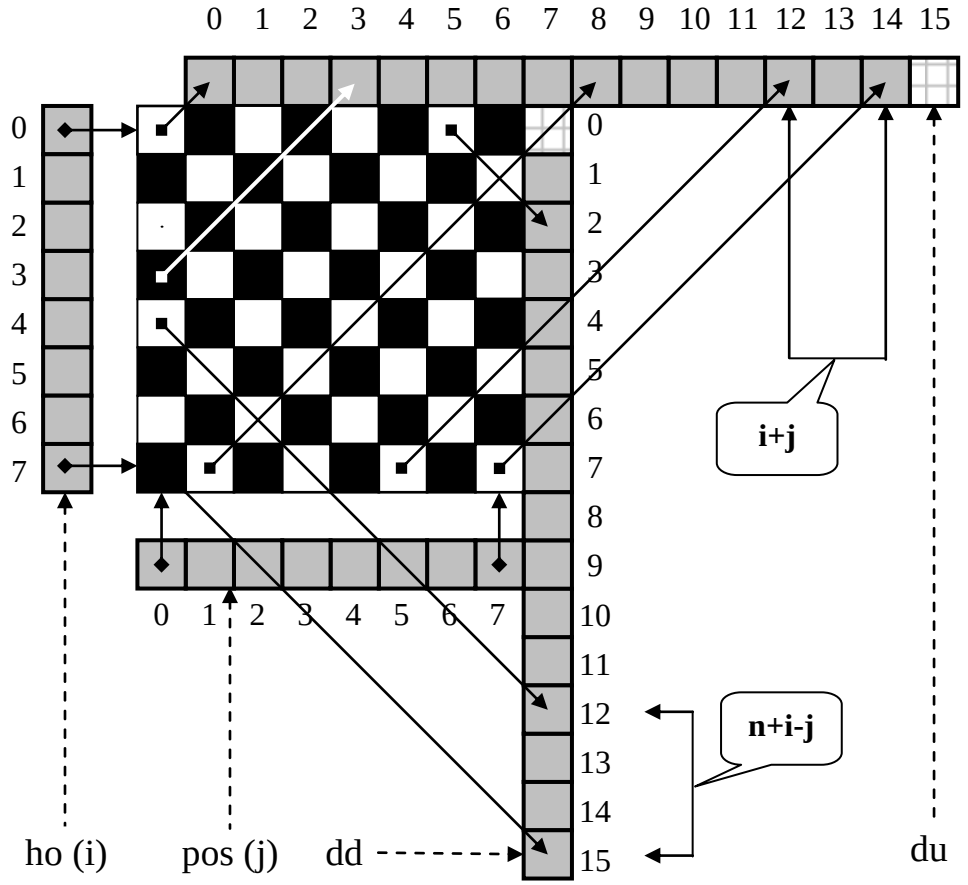
Проведем параметризацию задачи. Введем четыре вспомогательных вектора: pos, ho, dd и du с длинами n, n, $2n-1$ и $2n-1$ соответственно. Использовать их будем следующим образом (рис. 1):

- $ho_i = 1$, если на горизонтали с номером i $(i = 0, 1, \ldots n-1)$ имеется ферзь, и $ho_i = 0$ – в противном случае;
- $du_s = 1$, если на диагонали с номером s $(s = 0, 1, \ldots 2n-2)$, идущей слева направо и снизу вверх, имеется ферзь, и $du_s = 0$ – в противном случае;
- $dd_s = 1$, если на диагонали с номером s $(s = 0, 1, \ldots 2n-1)$, идущей слева направо и сверху вниз, имеется ферзь, и $dd_{s+1} = 0$ – в противном случае;
- $pos_j = i$, если в позиции (i, j) $(i, j = 0, 1, \ldots n-1)$ стоит ферзь.

Использование этих соглашений позволяет получить такие утверждения:

- В позицию (i, j) можно поставить ферзь, если $ho_i + du_{i+j} + dd_{n+i-j} = 0$.
- Поставить ферзь в позицию (i, j) равносильно присваиваниям: $ho_i = 1$, $du_{i+j} = 1$, $dd_{n+i-j} = 1$.

- Убрать ферзь из позиции (i, j) равносильно присваиваниям: $ho_i = 0$, $du_{i+j} = 0$, $dd_{n+i-j} = 0$.



Данное описание алгоритма является моделью решения общей задачи о нахождении всех вариантов расстановок. Рекурсия здесь осуществляется по не совсем стандартной схеме. В каждом рекурсивном вызове глубины j делается попытка поместить ферзь в некоторую позицию i столбца j $(i, j = 0, 1, \ldots n-1)$, а сам вызов соответствует переходу от работы с текущим столбцом к работе со следующим столбцом. При этом в начале вычислений и при переходах к любому последующему рекурсивному вызову параметр i меняется от нуля и далее с шагом, равным единице, пытаясь принять значение наименьшего номера поля, допустимого для установки ферзя. При переходах к любому предыдущему рекурсивному вызову параметр i продолжает изменяться от своего текущего на данном уровне значения с шагом, равным единице, также пытаясь принять значение наименьшего номера поля, допустимого для установки ферзя. Если в текущем столбце ферзь установить уже не удается, то создавшуюся ситуацию назовем тупиком. Попадание в тупик приводит к завершению текущего рекурсивного вызова, то есть к возврату к предыдущему столбцу и продолжению работы с ним. Иных случаев завершения рекурсивных вызовов не существует. Поэтому базой рекурсии мы должны считать совокупность всех тупиков. Заметим, что в данном случае элементы базы заранее до вычислений неизвестны.

После установки ферзя в одну из строк i последнего столбца $j = n-1$ формируется одно из решений задачи – при поиске одного варианта расстановки на этом этапе следует завершить выполнение алгоритма. При поиске всех расстановок вычисления прекращаются, когда мы попадаем в тупик при работе со столбцом 0. Полученные решения задачи, если они есть, возвращаются в виде столбцов матрицы otv, начиная от первого и далее.

Рассмотрим, как реализуется декомпозиция. Для этого вместо исходной задачи удобно решать ее следующее обобщение.

На доске размера n×m $(m = n, n-1, \ldots, 0)$ требуется установить m ферзей так, чтобы они не били друг друга. При этом имеются некоторые клетки доски, на которые ферзь заведомо

ставить нельзя. Множество этих «запретных» клеток обозначим через ω.

Исходная задача есть E (n, n, $\varnothing$). Проведем ее декомпозицию. Представим доску в виде двух частей: нулевого столбца (A) и оставшейся части (B). Соответственно этому разбиению будем решать задачи E (n, 1, $\varnothing$) и E (n, n–1, ω). Каждое из n возможных решений i (ферзь установлен в строке $i = 0, 1, \ldots n-1$) первой задачи однозначно определяет множество $\omega = \omega(i)$ запретных клеток для второй задачи. При этом в $\omega(i)$ попадают те клетки B, которые в «объединенной» доске бьет ферзь, установленный в строке i доски A. Пусть i зафиксировано и найдено множество $d(i)$ решений второй задачи E (n, n–1, ω(i)) для доски B. Тогда расположение ферзя в строке i доски A и любое из полученных решений $x \in d(i)$ на B в совокупности дают различные решения otv(i) исходной задачи E (n, n, $\varnothing$). Для получения всех решений этой задачи остается лишь взять объединение по i множеств otv(i).

При анализе трудоемкости алгоритма получаем, что глубина рекурсии равна n^2 – при каждом рекурсивном вызове по j ($j = 0, 1, \ldots n-1$) происходит n рекурсивных вызовов по i ($i = 0, 1, \ldots n-1$).

Задание:

1. Составьте программу для примера 1.
2. Составьте рекурсивную функцию, находящую количество возможных расстановок n ферзей на шахматной доске размером n×n так, чтобы они не били друг друга.

Содержание отчета

– Название работы;
– Цель работы;
– Формулировку задания;
– Решенные задания, с пояснением решения в виде таблиц, рассуждений, рисунков.

Контрольные вопросы

1. В чем состоит методом перебора с возвратом?
2. Разъясните вычислительную схему метода перебора с возвратом.

Список литературы:

Основная литература

1. Окулов С. М. Программирование в алгоритмах. / М. Окулов. – М.: БИНОМ. Лаборатория знаний, 2002. – 341 с.
2. Иванов Б. Н. Дискретная математика. Алгоритмы и программы. – М.: Лаборатория Базовых Знаний, 2002.

Дополнительная литература

1. Семакин И. Г. Основы программирования: учебник для сред. проф. образования / И. Г. Семакин, А. П. Шестаков. – М.: Издательский центр «Академия», 2007.
2. Акимов О.Е. Дискретная математика: логика, группы, графы. – М.: Лаборатория Базовых Знаний, 2001.

Практическая работа № 6

Тема: Реализация метода перебора в задачах поиска.

Цель: закрепление умений по программированию алгоритмов.

Пояснения к работе:

Одним из важнейших действий со структурированной информацией является поиск. Поиск – процесс нахождения конкретной информации в ранее созданном множестве данных. Обычно данные представляют собой записи, каждая из которых имеет хотя бы один ключ. Ключ поиска – это поле записи, по значению которого происходит поиск. Ключи используются для отличия одних записей от других. Целью поиска является нахождение всех записей (если они есть) с

данным значением ключа.

Структуру данных, в которой проводится поиск, можно рассматривать как таблицу символов (таблицу имен или таблицу идентификаторов) – структуру, содержащую ключи и данные, и допускающую две операции – вставку нового элемента и возврат элемента с заданным ключом. Иногда таблицы символов называют словарями по аналогии с хорошо известной системой упорядочивания слов в алфавитном порядке: слово – ключ, его толкование – данные.

Поиск является одним из наиболее часто встречаемых действий в программировании. Существует множество различных алгоритмов поиска, которые принципиально зависят от способа организации данных. У каждого алгоритма поиска есть свои преимущества и недостатки. Поэтому важно выбрать тот алгоритм, который лучше всего подходит для решения конкретной задачи.

Поставим задачу поиска в линейных структурах. Пусть задано множество данных, которое описывается как массив, состоящий из некоторого количества элементов. Проверим, входит ли заданный ключ в данный массив. Если входит, то найдем номер этого элемента массива, то есть, определим первое вхождение заданного ключа (элемента) в исходном массиве.

Таким образом, определим общий алгоритм поиска данных:

Шаг 1. Вычисление элемента, что часто предполагает получение значения элемента, ключа элемента и т.д.

Шаг 2. Сравнение элемента с эталоном или сравнение двух элементов (в зависимости от постановки задачи).

Шаг 3. Перебор элементов множества, то есть прохождение по элементам массива.

Основные идеи различных алгоритмов поиска сосредоточены в методах перебора и стратегии поиска.

Последовательный (линейный) поиск

Последовательный (линейный) поиск – это простейший вид поиска заданного элемента на некотором множестве, осуществляемый путем последовательного сравнения очередного рассматриваемого значения с искомым до тех пор, пока эти значения не совпадут.

Идея этого метода заключается в следующем. Множество элементов просматривается последовательно в некотором порядке, гарантирующем, что будут просмотрены все элементы множества (например, слева направо). Если в ходе просмотра множества будет найден искомый элемент, просмотр прекращается с положительным результатом; если же будет просмотрено все множество, а элемент не будет найден, алгоритм должен выдать отрицательный результат.

Алгоритм последовательного поиска

Шаг 1. Полагаем, что значение переменной цикла $i = 0$.

Шаг 2. Если значение элемента массива $x[i]$ равно значению ключа key, то возвращаем значение, равное номеру искомого элемента, и алгоритм завершает работу. В противном случае значение переменной цикла увеличивается на единицу $i = i + 1$.

Шаг 3. Если $i < k$, где k – число элементов массива x, то выполняется Шаг 2, в противном случае – работа алгоритма завершена и возвращается значение равное -1.

При наличии в массиве нескольких элементов со значением key данный алгоритм находит только первый из них (с наименьшим индексом).

```
//описание функции последовательного поиска
int LinearSearch(int *x, int k, int key){
  int i = 0;
  for ( i = 0 ; i < k ; i++ )
    if ( x[i] == key )
      break;
  return i < k ? i : -1;
}
```

Время выполнения данного алгоритма поиска для вещественных чисел n/ε, где n – количество элементов множества, а ε – точность. Поиск на дискретном множестве из n элементов осуществляется в худшем случае за n итераций, а в среднем этот алгоритм требует $n/2$ итераций цикла. Следовательно, временная сложность последовательного поиска пропорциональна $O(n)$. Никаких ограничений на порядок элементов в массиве данный алгоритм не накладывает.

Недостатком рассматриваемого алгоритма поиска является то, что в худшем случае осуществляется просмотр всего массива. Поэтому данный алгоритм используется, если множество

содержит небольшое количество элементов.

Достоинства последовательного поиска заключаются в том, что он прост в реализации, не требует сортировки значений множества, дополнительной памяти и дополнительного анализа функций. Следовательно, может работать в потоковом режиме при непосредственном получении данных из любого источника.

Существует модификация алгоритма последовательного поиска, которая ускоряет поиск. Эта модификация является небольшим усовершенствованием рассмотренного алгоритма поиска.

Идея поиска с барьером состоит в том, чтобы не проверять каждый раз в цикле условие, связанное с границами множества. Это можно обеспечить, установив в данном множестве так называемый барьер. Под барьером понимается любой элемент, который удовлетворяет условию поиска. Тем самым будет ограничено изменение индекса.

Выход из цикла, в котором теперь остается только условие поиска, может произойти либо на найденном элементе, либо на барьере. Существует два способа установки барьера: дополнительным элементом или вместо крайнего элемента массива.

```
//описание функции последовательного поиска с барьером
int LinearSearchWithBarrier(int *x, int k, int key){
    x = (int *)realloc(x,(k+1)*sizeof(int));
    x[k] = key;
    int i = 0;
    while ( x[i] != key )
        i++;
    return i < k ? i : -1;
}
```

Заметим, что поиск с барьером работает быстрее, но временная сложность алгоритма остается такой же $O(n)$, где n – количество элементов множества. Гораздо больший интерес представляют методы, не только работающие быстро, но и реализующие алгоритмы с меньшей сложностью.

Бинарный (двоичный) поиск

Бинарный (двоичный, дихотомический) поиск – это поиск заданного элемента на упорядоченном множестве, осуществляемый путем неоднократного деления этого множества на две части таким образом, что искомый элемент попадает в одну из этих частей. Поиск заканчивается при совпадении искомого элемента с элементом, который является границей между частями множества или при отсутствии искомого элемента.

Бинарный поиск применяется к отсортированным множествам и заключается в последовательном разбиении множества пополам и поиска элемента только в одной половине на каждой итерации.

Таким образом, идея этого метода заключается в следующем. Поиск нужного значения среди элементов упорядоченного массива (по возрастанию или по убыванию) начинается с определения значения центрального элемента этого массива. Значение данного элемента сравнивается с искомым значением и в зависимости от результатов сравнения предпринимаются определенные действия. Если искомое и центральное значения оказываются равны, то поиск завершается успешно. Если искомое значение меньше центрального или больше, то формируется массив, состоящий из элементов, находящихся слева или справа от центрального соответственно. Затем поиск повторяется в новом массиве (рис).

Алгоритм бинарного поиска

Шаг 1. Определить номер среднего элемента массива

$$middle = (high + low)/2.$$

Шаг 2. Если значение среднего элемента массива равно искомому, то возвращаем значение, равное номеру искомого элемента, и алгоритм завершает работу.

Шаг 3. Если искомое значение больше значения среднего элемента, то возьмем в качестве массива все элементы справа от среднего, иначе возьмем в качестве массива все элементы слева от среднего (в зависимости от характера упорядоченности). Перейдем к Шагу 1.

В массиве может встречаться несколько элементов со значениями, равными ключу. Данный алгоритм находит первый совпавший с ключом элемент, который в порядке следования в массиве может быть ни первым, ни последним среди равных ключу. Например, в массиве чисел 1, 5, 5, 5, 5, 5, 5, 7, 8 с ключом key=5 совпадет элемент с порядковым номером 4, который не относится ни к первому, ни к последнему.

Существуют две модификации рассматриваемого алгоритма для поиска первого и

последнего вхождения. Все зависит от того, как выбирается средний элемент: округлением в меньшую или большую сторону. В первом случае средний элемент относится к левой части массива, а во втором – к правой.

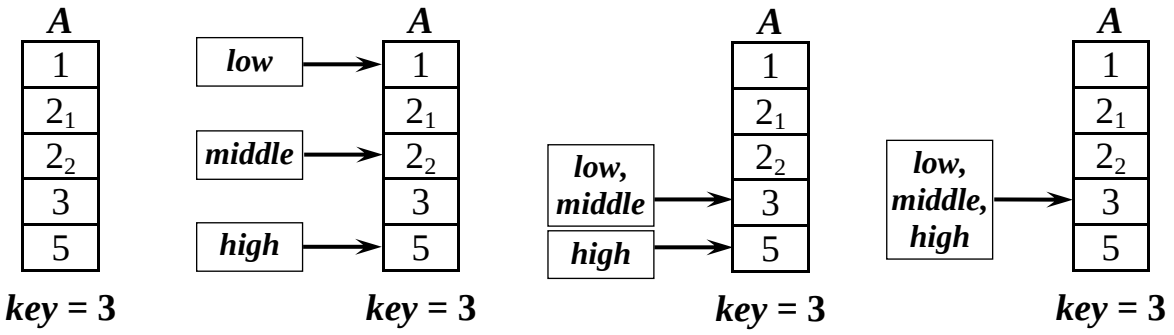


Рис. Демонстрация алгоритма бинарного поиска

```
//описание функции бинарного поиска
int BinarySearch(int *x, int k, int key){
  bool found = false;
  int high = k - 1, low = 0;
  int middle = (high + low) / 2;
  while ( !found && high >= low ){
    if (key == x[middle])
      found = true;
    else if (key < x[middle])
      high = middle - 1;
    else
      low = middle + 1;
      middle = (high + low) / 2;
  }
  return found ? middle : -1 ;
}
```

В процессе работы алгоритма бинарного поиска размер фрагмента, где этот поиск должен продолжаться, каждый раз уменьшается примерно в два раза. Это обеспечивает сложность алгоритма пропорциональную $O(\log n)$, где n – количество элементов множества.

Время выполнения алгоритма бинарного поиска: если функция имеет вещественный аргумент, найти решение с точностью до ε можно за время $\log\frac{1}{\varepsilon}$, а если аргумент дискретен, то поиск решения займет $(1+\log n)$ времени.

Достоинством данного алгоритма является относительная быстрота выполнения поиска, по сравнению с алгоритмом последовательного поиска. Недостаток заключается в том, что бинарный поиск может применяться только на упорядоченном множестве.

Задание:

1. На основании приведенных в лекции функций реализуйте алгоритмы последовательного и бинарного поиска.
2. В связи с визитом Императора Палпатина было решено обновить состав дроидов в ангаре 32. Из-за кризиса было решено новых дроидов не закупать, но выкинуть пару старых. Как известно, Палпатин не переносит дроидов с маленькими серийными номерами, так что все, что требуется – найти среди них двух, у которых серийные номера наименьшие.

Формат входного файла

Первая строка входного файла содержит целое число N – количество дроидов. ($2 \le N \le 1000$), вторая строка – N целых чисел, по модулю не превышающих $2\cdot 10^9$ – номера дроидов.

Формат выходного файла

Выведите два числа: первым – последний по величине из номеров дроидов (такого следует утилизировать в первую очередь), а вторым – предпоследний.

Пример входного файла

```
5
49 100 23 -100 157
```

Пример выходного файла

```
-100 23
```

Пример входного файла

```
4
99 1 5 1
```

Пример выходного файла

```
1 1
```

3. Некто загадал число от 1 до N. За какое наименьшее количество вопросов (на которые он отвечает «да» или «нет») можно угадать задуманное число?

Формат входных данных

Вводится одно число N ($1 < N < 10001$).

Формат выходных данных

Выведите наименьшее количество вопросов, которого гарантированно хватит, чтобы угадать задуманное число.

Пример входного файла

```
6
```

Пример выходного файла

```
3
```

4. Задана матрица K, содержащая n строк и m столбцов. Седловой точкой этой матрицы назовем элемент, который одновременно является минимумом в своей строке и максимумом в своем столбце. Найдите количество седловых точек заданной матрицы.

Формат входного файла

Первая строка входного файла содержит целые числа n и m ($1 \le n, m \le 750$). Далее следуют n строк по m чисел в каждой. j-ое число i-ой строки равно k_{ij}. Все k_{ij} по модулю не превосходят 1000.

Формат выходного файла

В выходной файл выведите ответ на задачу.

Пример входного файла

```
2 2
0 0
0 0
```

Пример выходного файла

```
4
```

Пример входного файла

```
2 2
1 2
3 4
```

Пример выходного файла

```
1
```

5. Спортсмен Василий участвовал в соревнованиях по хоккейболу и получил в личном зачете серебряную медаль. Известно, что участники, получившие одинаковое количество очков, награждаются одинаковыми наградами. Известно, что были разыграны золотые серебряные и бронзовые медали. В задаче не спрашиваются правила хоккейбола. Необходимо только определить сколько очков набрал Василий. Для решения данной задачи массив лучше не использовать.

Формат входного файла

На первой строке дано число N ($2 \le N \le 1000$) количество спортсменов, участвовавших в соревнованиях, на второй N целых чисел – результаты через пробел.

Формат выходного файла

Требуется вывести одно число – результат Василия.

Пример входного файла

```
5
4 3 3 1 2
```

Пример выходного файла

```
3
```

Пример входного файла

```
8
1 2 5 3 5 1 1 6
```

Пример выходного файла

```
5
```

Содержание отчета

– Название работы;
– Цель работы;
– Формулировку задания;
– Решенные задания, с пояснением решения в виде таблиц, рассуждений, рисунков.

Контрольные вопросы

1. Чем можно объяснить многообразие алгоритмов поиска в линейных структурах?
2. В чем преимущества поиска с барьером по сравнению с последовательным поиском?
3. Нахождение какого по порядку элемента в линейном множестве (первого, последнего) гарантирует алгоритм прямого поиска? Как в этом случае должен быть выполнен просмотр?

Список литературы:

Основная литература

1. Окулов С. М. Программирование в алгоритмах. / М. Окулов. – М.: БИНОМ. Лаборатория знаний, 2002. – 341 с.
2. Иванов Б. Н. Дискретная математика. Алгоритмы и программы. – М.: Лаборатория Базовых Знаний, 2002.

Дополнительная литература

1. Семакин И. Г. Основы программирования: учебник для сред. проф. образования / И. Г. Семакин, А. П. Шестаков. – М.: Издательский центр «Академия», 2007.
2. Акимов О.Е. Дискретная математика: логика, группы, графы. – М.: Лаборатория Базовых Знаний, 2001.

Практическая работа № 7

Тема: Реализация алгоритма сортировки вставками.

Цель: закрепление умений по программированию алгоритмов.

Пояснения к работе:

Сортировка вставками – простой алгоритм сортировки, преимущественно использующийся в учебном программировании. К положительной стороне метода относится простота реализации, а также его эффективность на частично упорядоченных последовательностях, и/или состоящих из небольшого числа элементов. Тем не менее, высокая вычислительная сложность не позволяет рекомендовать алгоритм в повсеместном использовании.

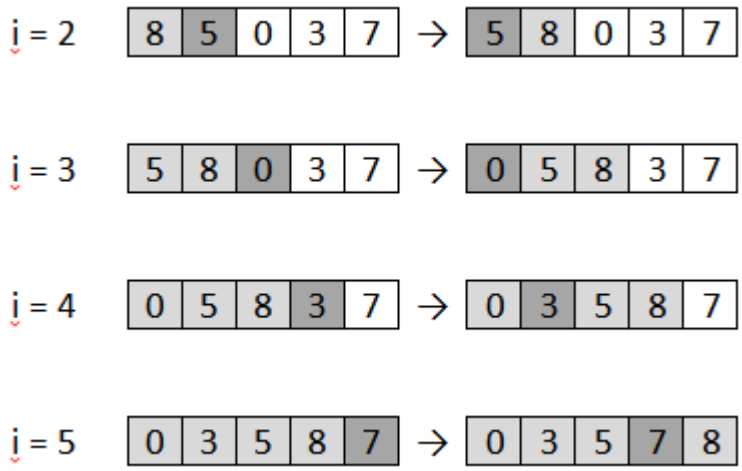
Рассмотрим алгоритм сортировки вставками на примере колоды игральных карт. Процесс их упорядочивания по возрастанию (в колоде карты расположены в случайном порядке) будет следующим. Обратим внимание на вторую карту, если ее значение меньше первой, то меняем эти карты местами, в противном случае карты сохраняют свои позиции, и алгоритм переходит к шагу 2. На 2-ом шаге смотрим на третью карту, здесь возможны четыре случая отношения значений карт:

1. первая и вторая карта меньше третьей;
2. первая и вторая карта больше третьей;
3. первая карта уступает значением третьей, а вторая превосходит ее;
4. первая карта превосходит значением третью карту, а вторая уступает ей.

В первом случае не происходит никаких перестановок. Во втором – вторая карта

смещается на место третьей, первая на место второй, а третья карта занимает позицию первой. В предпоследнем случае первая карта остается на своем месте, в то время как вторая и третья меняются местами. Ну и наконец, последний случай требует рокировки лишь первой и третьей карт. Все последующие шаги полностью аналогичны расписанным выше.

Рассмотрим на примере числовой последовательности процесс сортировки методом вставок. Клетка, выделенная темно-серым цветом – активный на данном шаге элемент, ему также соответствует i-ый номер. Светло-серые клетки это те элементы, значения которых сравниваются с i-ым элементом. Все, что закрашено белым – не затрагиваемая на шаге часть последовательности.



Таким образом, получается, что на каждом этапе выполнения алгоритма сортируется некоторая часть массива, размер которой с шагом увеличивается, и в конце сортируется весь массив целиком.

Задание:

1. Запрограммируйте предложенный алгоритм.
2. Выполнить сортировку только четных элементов массива (нечетные элементы остаются на своих местах)
3. Выполнить сортировку элементов, записанных на нечетных местах.

Содержание отчета

– Название работы;
– Цель работы;
– Формулировку задания;
– Решенные задания, с пояснением решения в виде таблиц, рассуждений, рисунков.

Контрольные вопросы

1. В чем состоит метод сортировки выбором?
2. Продемонстрируйте метод на примере одномерного массива из 5 элементов.

Список литературы:

Основная литература

1. Окулов С. М. Программирование в алгоритмах. / М. Окулов. – М.: БИНОМ. Лаборатория знаний, 2002. – 341 с.
2. Иванов Б. Н. Дискретная математика. Алгоритмы и программы. – М.: Лаборатория Базовых Знаний, 2002.

Дополнительная литература

1. Семакин И. Г. Основы программирования: учебник для сред. проф. образования / И. Г. Семакин, А. П. Шестаков. – М.: Издательский центр «Академия», 2007.
2. Акимов О.Е. Дискретная математика: логика, группы, графы. – М.: Лаборатория Базовых Знаний, 2001.

Практическая работа № 8

Тема: Реализация алгоритма пузырьковой сортировки.

Цель: закрепление умений по программированию алгоритмов.

Пояснения к работе:

Задание:

При работе с массивами данных не редко возникает задача их **сортировки по возрастанию или убыванию, т.е. упорядочивания**. Это значит, что элементы того же нужно расположить строго по порядку. Например, в случае сортировки по возрастанию предшествующий элемент должен быть меньше последующего (или равен ему).

Алгоритм решения задачи:

Существует множество методов сортировки. Одни из них являются более эффективными, другие – проще для понимания. Достаточно простой для понимания является сортировка **методом пузырька**, который также называют **методом простого обмена**. В чем же он заключается, и почему у него такое странное название: "метод пузырька"?

Как известно воздух легче воды, поэтому пузырьки воздуха всплывают. Это просто аналогия. В сортировке методом пузырька по возрастанию более легкие (с меньшим значением) элементы постепенно "всплывают" в начало массива, а более тяжелые друг за другом опускаются на дно (в конец массива).

Алгоритм и особенности этой сортировки таковы:

При первом проходе по массиву элементы попарно сравниваются между собой: первый со вторым, затем второй с третьим, следом третий с четвертым и т.д. Если предшествующий элемент оказывается больше последующего, то их меняют местами.

Не трудно догадаться, что постепенно самое большое число оказывается последним. Остальная часть массива остается не отсортированной, хотя некоторое перемещение элементов с меньшим значением в начало массива наблюдается.

При втором проходе незачем сравнивать последний элемент с предпоследним. Последний элемент уже стоит на своем месте. Значит, число сравнений будет на одно меньше.

На третьем проходе уже не надо сравнивать предпоследний и третий элемент с конца. Поэтому число сравнений будет на два меньше, чем при первом проходе.

В конце концов, при проходе по массиву, когда остаются только два элемента, которые надо сравнить, выполняется только одно сравнение.

После этого первый элемент не с чем сравнивать, и, следовательно, последний проход по массиву не нужен. Другими словами, количество проходов по массиву равно m-1, где m – это количество элементов массива.

Количество сравнений в каждом проходе равно m-i, где i – это номер прохода по массиву (первый, второй, третий и т.д.).

При обмене элементов массива обычно используется "буферная" (третья) переменная, куда временно помещается значение одного из элементов.

Задание:

1. Запрограммируйте предложенный алгоритм.
2. Выполнить сортировку только нечетных элементов массива (четные элементы остаются на своих местах)
3. Выполнить сортировку элементов, записанных на четных местах.

Содержание отчета

– Название работы;
– Цель работы;
– Формулировку задания;
– Решенные задания, с пояснением решения в виде таблиц, рассуждений, рисунков.

Контрольные вопросы

1. В чем состоит метод сортировки пузырьком?
2. За счет чего в алгоритме происходят операций сравнения и перестановок?

Список литературы:

Основная литература

1. Окулов С. М. Программирование в алгоритмах. / М. Окулов. – М.: БИНОМ. Лаборатория знаний, 2002. – 341 с.
2. Иванов Б. Н. Дискретная математика. Алгоритмы и программы. – М.: Лаборатория Базовых Знаний, 2002.

Дополнительная литература

1. Семакин И. Г. Основы программирования: учебник для сред. проф. образования / И. Г. Семакин, А. П. Шестаков. – М.: Издательский центр «Академия», 2007.
2. Акимов О.Е. Дискретная математика: логика, группы, графы. – М.: Лаборатория Базовых Знаний, 2001.

Интернет ресурсы

http://edunow.su/site/content/algorithms/sortirovka_massiva

Практическая работа № 9

Тема: Реализация алгоритма сортировки выбором.

Цель: закрепление умений по программированию алгоритмов.

Пояснения к работе:

Является одним из самых простых алгоритмов сортировки массива. Смысл в том, чтобы идти по массиву и каждый раз искать минимальный элемент массива, обменивая его с начальным элементом неотсортированной части массива. На небольших массивах может оказаться даже эффективнее, чем более сложные алгоритмы сортировки, но в любом случае проигрывает на больших массивах. Число обменов элементов по сравнению с "пузырьковым" алгоритмом N/2, где N - число элементов массива.

Алгоритм:

1. Находим минимальный элемент в массиве.
2. Меняем местами минимальный и первый элемент местами.
3. Опять ищем минимальный элемент в неотсортированной части массива
4. Меняем местами уже второй элемент массива и минимальный найденный, потому как первый элемент массива является отсортированной частью.
5. Ищем минимальные значения и меняем местами элементы,пока массив не будет отсортирован до конца.

Задание:

1. Запрограммируйте предложенный алгоритм.
2. Выполните сортировку элементов массива кратных 5 (остальные элементы остаются на своих местах)
3. Выполните сортировку элементов, записанных на четных местах.

Содержание отчета

– Название работы;
– Цель работы;
– Формулировку задания;
– Решенные задания, с пояснением решения в виде таблиц, рассуждений, рисунков.

Контрольные вопросы

1. В чем состоит метод сортировки выбором?
2. За счет чего в алгоритме происходит выигрыш при выполнении операций сравнения и перестановок?

Список литературы:

Основная литература

1. Окулов С. М. Программирование в алгоритмах. / М. Окулов. – М.: БИНОМ. Лаборатория знаний, 2002. – 341 с.
2. Иванов Б. Н. Дискретная математика. Алгоритмы и программы. – М.: Лаборатория Базовых Знаний, 2002.

Дополнительная литература

1. Семакин И. Г. Основы программирования: учебник для сред. проф. образования / И. Г. Семакин, А. П. Шестаков. – М.: Издательский центр «Академия», 2007.

2. Акимов О.Е. Дискретная математика: логика, группы, графы. – М.: Лаборатория Базовых Знаний, 2001.

Интернет ресурсы

http://edunow.su/site/content/algorithms/sortirovka_massiva

Практическая работа № 10

Тема: Реализация алгоритма быстрой сортировки.

Цель: закрепление умений по программированию алгоритмов.

Пояснения к работе:

Метод быстрой сортировки был впервые описан Ч.А.Р. Хоаром в 1962 году. Быстрая сортировка – это общее название ряда алгоритмов, которые отражают различные подходы к получению критичного параметра, влияющего на производительность метода.

При общем рассмотрении алгоритма быстрой сортировки, отметим, что этот метод основывается на последовательном разделении сортируемого набора данных на блоки меньшего размера таким образом, что между значениями разных блоков обеспечивается отношение упорядоченности (для любой пары блоков все значения одного из этих блоков не превышают значений другого блока).

Опорным (ведущим) элементом называется некоторый элемент массива, который выбирается определенный образом. С точки зрения корректности алгоритма выбор опорного элемента безразличен. С точки зрения повышения эффективности алгоритма выбираться должна медиана, но без дополнительных сведений о сортируемых данных ее обычно невозможно получить. Необходимо выбирать постоянно один и тот же элемент (например, средний или последний по положению) или выбирать элемент со случайно выбранным индексом.

Алгоритм быстрой сортировки Хоара

Пусть дан массив $x[n]$ размерности n.

Шаг 1. Выбирается опорный элемент массива.

Шаг 2. Массив разбивается на два – левый и правый – относительно опорного элемента. Реорганизуем массив таким образом, чтобы все элементы, меньшие опорного элемента, оказались слева от него, а все элементы, большие опорного – справа от него.

Шаг 3. Далее повторяется шаг 2 для каждого из двух вновь образованных массивов. Каждый раз при повторении преобразования очередная часть массива разбивается на два меньших и т. д., пока не получится массив из двух элементов (рис).

Быстрая сортировка стала популярной прежде всего потому, что ее нетрудно реализовать, она хорошо работает на различных видах входных данных и во многих случаях требует меньше затрат ресурсов по сравнению с другими методами сортировки.

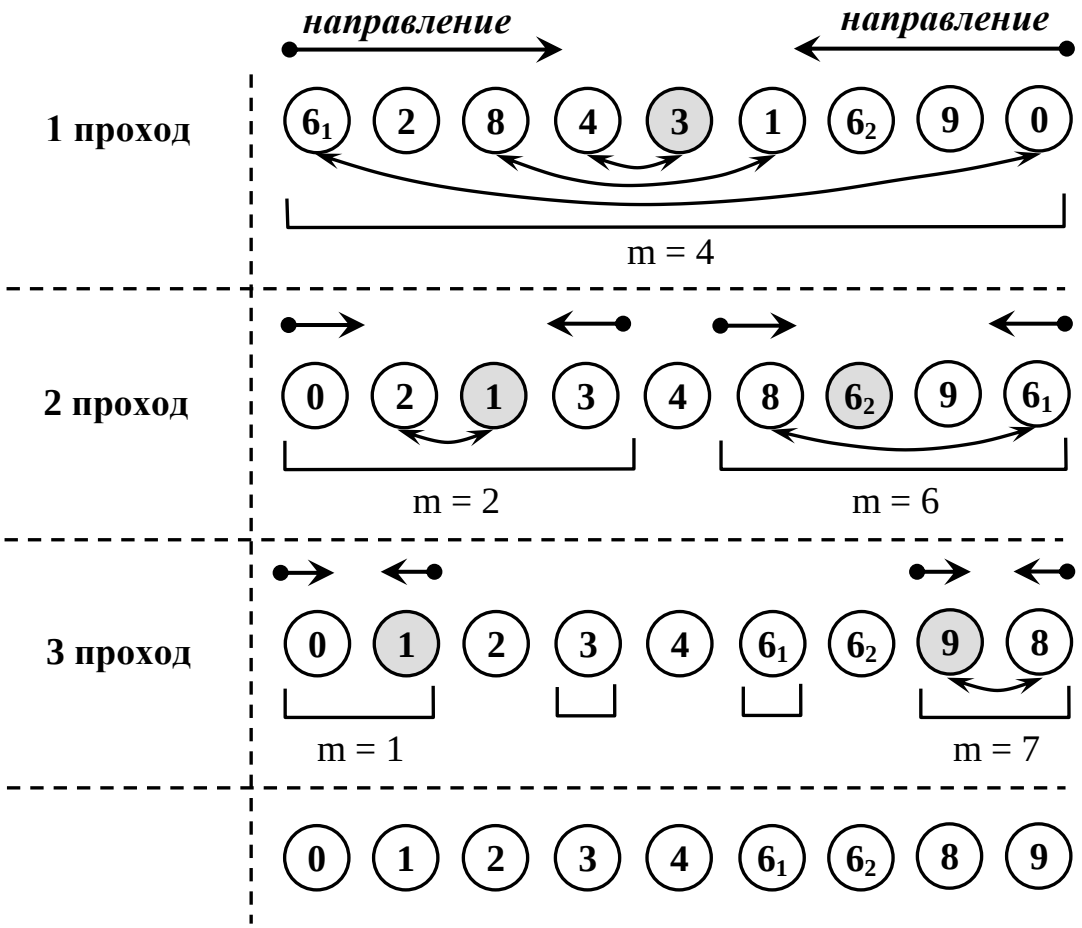
Выберем в качестве опорного элемент, расположенный на средней позиции.

Эффективность быстрой сортировки в значительной степени определяется правильностью выбора опорных (ведущих) элементов при формировании блоков. В худшем случае трудоемкость метода имеет ту же сложность, что и пузырьковая сортировка, то есть порядка $O(n^2)$. При оптимальном выборе ведущих элементов, когда разделение каждого блока происходит на равные по размеру части, трудоемкость алгоритма совпадает с быстродействием наиболее эффективных способов сортировки, то есть порядка $O(n \log n)$. В среднем случае количество операций, выполняемых алгоритмом быстрой сортировки, определяется выражением $T(n) = O(1.4\, n \log n)$

Быстрая сортировка является наиболее эффективным алгоритмом из всех известных методов сортировки, но все усовершенствованные методы имеют один общий недостаток – невысокую скорость работы при малых значениях n.

Рекурсивная реализация быстрой сортировки позволяет устранить этот недостаток путем включения прямого метода сортировки для частей массива с небольшим количеством элементов. Анализ вычислительной сложности таких алгоритмов показывает, что если подмассив имеет девять или менее элементов, то целесообразно использовать прямой метод (сортировку простыми

вставками).



Сортировка слиянием

Алгоритм сортировки слиянием был изобретен Джоном фон Нейманом в 1945 году. Он является одним из самых быстрых способов сортировки.

Слияние – это объединение двух или более упорядоченных массивов в один упорядоченный.

Сортировка слиянием является одним из самых простых алгоритмов сортировки (среди быстрых алгоритмов). Особенностью этого алгоритма является то, что он работает с элементами массива преимущественно последовательно, благодаря чему именно этот алгоритм используется при сортировке в системах с различными аппаратными ограничениями (например, при сортировке данных на жестком диске). Кроме того, сортировка слиянием является алгоритмом, который может быть эффективно использован для сортировки таких структур данных, как связанные списки.

Данный алгоритм применяется тогда, когда есть возможность использовать для хранения промежуточных результатов память, сравнимую с размером исходного массива. Он построен на принципе «разделяй и властвуй». Сначала задача разбивается на несколько подзадач меньшего размера. Затем эти задачи решаются с помощью рекурсивного вызова или непосредственно, если их размер достаточно мал. Далее их решения комбинируются, и получается решение исходной задачи (рис).

Процедура слияния требует два отсортированных массива. Заметим, что массив из одного элемента по определению является отсортированным.

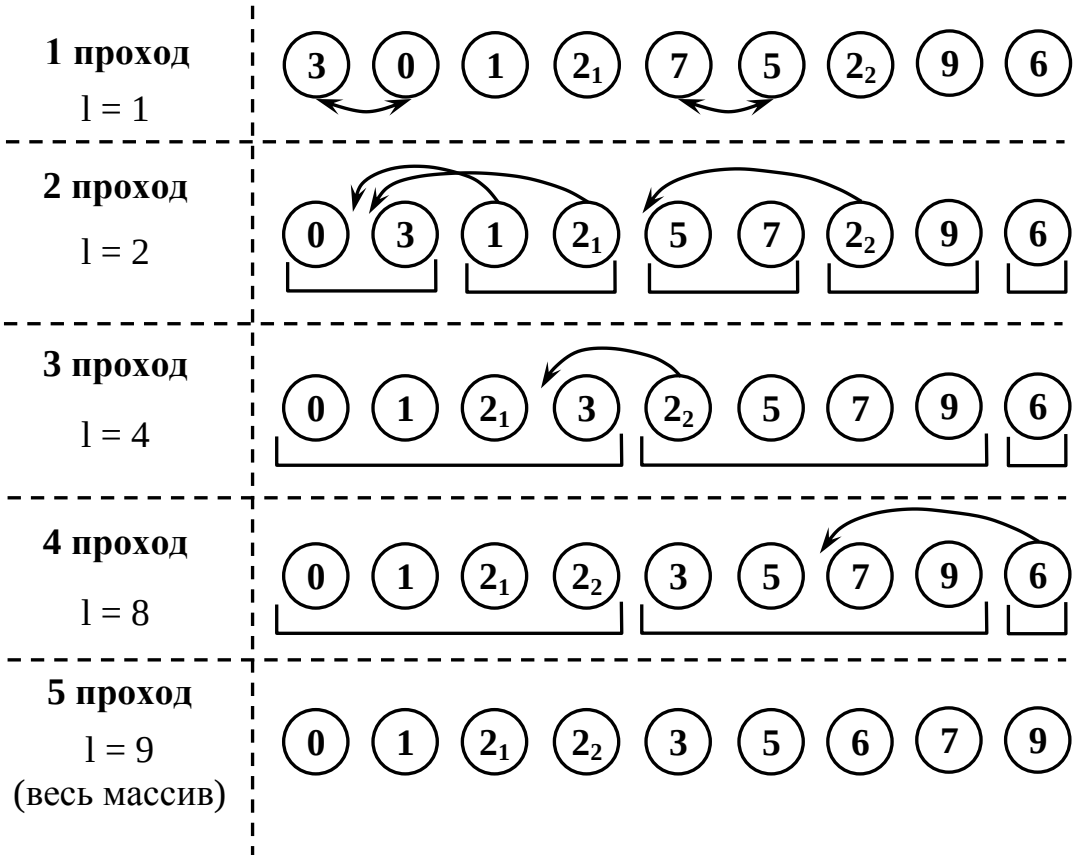
Алгоритм сортировки слиянием

Шаг 1. Разбить имеющиеся элементы массива на пары и осуществить слияние элементов каждой пары, получив отсортированные цепочки длины 2 (кроме, быть может, одного элемента, для которого не нашлось пары).

Шаг 2. Разбить имеющиеся отсортированные цепочки на пары, и осуществить слияние цепочек каждой пары.

Шаг 3. Если число отсортированных цепочек больше единицы, перейти к шагу 2.

Недостаток алгоритма заключается в том, что он требует дополнительную память размером порядка n (для хранения вспомогательного массива). Кроме того, он не гарантирует сохранение порядка элементов с одинаковыми значениями. Но его временная сложность всегда пропорциональна $O(n \log n)$.



Задание:

1. Запрограммируйте предложенные алгоритмы.
2. Выполните сортировку только положительных элементов массива (отрицательные элементы остаются на своих местах)
3. Выполните сортировку элементов, записанных на местах, кратных 3.

Содержание отчета

– Название работы;
– Цель работы;
– Формулировку задания;
– Решенные задания, с пояснением решения в виде таблиц, рассуждений, рисунков.

Контрольные вопросы

1. За счет чего в алгоритмах быстрых сортировок происходит выигрыш при выполнении операций сравнения и перестановок?
2. Какие из перечисленных алгоритмов наиболее эффективны на почти отсортированных массивах: бинарная пирамидальная сортировка, сортировка слиянием, сортировка Шелла и сортировка Хоара? За счет чего происходит выигрыш?
3. Почему алгоритмы быстрых сортировок не дают большого выигрыша при малых размерах массивов?

Список литературы:

Основная литература

1. Окулов С. М. Программирование в алгоритмах. / М. Окулов. – М.: БИНОМ. Лаборатория знаний, 2002. – 341 с.
2. Иванов Б. Н. Дискретная математика. Алгоритмы и программы. – М.: Лаборатория Базовых Знаний, 2002.

Дополнительная литература

1. Семакин И. Г. Основы программирования: учебник для сред. проф. образования / И. Г. Семакин, А. П. Шестаков. – М.: Издательский центр «Академия», 2007.
2. Акимов О.Е. Дискретная математика: логика, группы, графы. – М.: Лаборатория Базовых Знаний, 2001.

Интернет ресурсы

http://edunow.su/site/content/algorithms/sortirovka_massiva

Практическая работа № 11

Тема: Реализация основных арифметических операций на многоразрядных целых числах.
Цель: закрепление умений по программированию алгоритмов.
Пояснения к работе:

Рассмотрим достаточно популярную в программировании задачу на работу с "длинными" числами. Реально с "астрономическими" или "микроскопическими" числами приходится сталкиваться не так уж и часто. Тем не менее, упражнения, рассматриваемые в этой публикации, могут послужить хорошей тренировкой в области программирования и занять достойное место в классах с углубленным изучением информатики или на кружках по программированию. Алгоритмы, представленные ниже, записаны на Turbo Pascal, версия 7.0. При желании или необходимости они могут легко быть адаптированы к любой другой программной среде.

Диапазон представления целых чисел (**Integer, Word, LongInt**) ограничен, о чем не раз уже говорилось (впрочем, для действительных величин это замечание тоже актуально). Поэтому при решении задач всегда приходится действовать с оглядкой, — как бы не допустить возникновения ошибки выхода за диапазон или переполнения. Например, вычисляя факториал ($n! = 1 * 2 * 3 * \ldots * n$), в диапазоне представления величин типа`Integer` удастся правильно получить только 7! = 5040, а в диапазоне представления типа `LongInt` — 12! = 479001600. Для больших значений, конечно, можно использовать действительные типы данных, но это уже не гарантирует точного результата. Поэтому полезно для получения точных значений при действиях с многозначными числами разработать другие способы представления таких чисел, алгоритмы выполнения арифметических и других операций, процедуры ввода и вывода результатов и т.д.

Покажем реализацию решения такого рода задач на примере умножения одного многозначного числа на другое. Именно эта арифметическая операция наиболее часто используется при решении других задач.

Наиболее естественным способом представления многозначного числа является запись каждого его разряда в виде отдельного элемента линейного массива (или списка, где память под цифру будет отводиться по мере надобности, в то время как в массиве приходится заранее задавать максимальное количество элементов в нем). Пусть (для удобства дальнейших действий) разряды "длинного" числа при записи в массив нумеруются с единицы, начиная с разряда единиц, т.е. цифра из разряда единиц — элемент массива с номером один, цифра из разряда десятков — элемент массива с номером два и т.д. Определим для работы с "длинными" неотрицательными числами тип данных:

```
Const MNax = 2000;
Type Digit = 0..9;
     DlChislo = Array[1..Nmax] Of Digit;
```

Для решения поставленной задачи необходимо уметь выполнять следующие действия:
1) ввод "длинного" числа;
2) собственно умножение двух "длинных" чисел;
3) вывод "длинного" числа;
4) определение количества цифр в записи числа.

Каждую из подзадач реализуем в виде отдельной подпрограммы. Начнем с ввода. Ввести большое число целесообразно в виде строки, а в дальнейшем преобразовать в массив цифр. В

процедуре учтен указанный выше способ размещения "длинного" числа в массиве, т.е. с точки зрения пользователя число записывается как бы в обратном порядке.

```
{Процедура преобразования длинного числа, записанного
 в виде строки, в массив цифр; переменная OK принимает значение True,
 если в записи числа нет посторонних символов, отличных от десятичных
 цифр, иначе — false}
 Procedure Translate(S : String; Var A : DlChislo; Var OK : Boolean);
 Var I : Word;
 Begin
     Zero(A); I := Length(S); OK := True;
     While (I >= 1) And OK Do
      Begin
          If S[I] In ['0'..'9']
          Then A[Length(S)- I + 1]:= Ord(S[I]) - 48
          Else OK := False; I := I - 1
      End
 End;
```

В процедуре вызывается подпрограмма `Zero(A)`, назначение которой — запись нуля в каждый разряд длинного числа. Вот текст этой процедуры:

```
{Процедура обнуления длинного числа}
Procedure Zero(Var A : DlChislo);
Var I : Integer;
Begin
   For I := 1 To NMax Do A[I] := 0;
End;
```

Таким образом, длинное число записано в массив, где впереди (в качестве элементов с большими номерами) стоят незначащие нули. При выполнении действий и выводе ответа они не учитываются.

Сейчас разработаем функцию определения количества значащих цифр в записи числа, поскольку она потребуется при реализации подпрограммы умножения.

```
{Функция определения количества цифр в записи длинного числа}
Function Dlina(C : DlChislo) : Integer;
Var I : Integer;
Begin
  I := NMax;
  While (I > 1) And (C[I] = 0) Do I := I - 1;
  Dlina := I
End;
```

При ее разработке было использовано следующее соображение: если число не равно нулю, то количество цифр в его записи равно номеру первой цифры, отличной от нуля, если просмотр числа осуществляется от старшего разряда к младшему. Если же длинное число равно нулю, то получается, что количество цифр в его записи равно одной, что и требовалось.

Ну и, наконец, главная процедура, ради которой и была проделана вся предшествующая работа. При составлении алгоритма используется идея умножения "столбиком", хотя в нашем варианте сложение выполняется не по окончанию умножения, а по ходу его, т.е. перемножив очередные цифры, сразу же добавляем результирующую цифру в нужный разряд и формируем перенос в следующий разряд.

```
{Процедура умножения длинных чисел.
 A, B — множители, C — произведение}
 Procedure Multiplication(A, B : DlChislo; Var C : DlChislo);
 Var I, J : Integer; P : Digit; VspRez : 0..99;
  Begin
    Zero(C);
    For I := 1 To Dlina(A) Do {Цикл по количеству цифр
                               в первом числе}
     Begin
```

```
            P := 0; {Первоначально перенос равен нулю}
            For J := 1 To Dlina(B) Do {Цикл по количеству цифр
                                       во втором числе}
             Begin
               VspRez := A[I] * B[J] + P + C[I + J - 1];
               C[I + J - 1] := VspRez Mod 10; {Очередное значение
цифры в
                                                разряде I + J - 1}
               P := VspRez Div 10 {Перенос в следующий разряд}
             End;
           C[I + J] := P {последний перенос может быть отличен от
нуля,
                          запишем его в пока ещё свободный разряд}
         End
       End;
```

Сейчас приведем листинг программы целиком.

```
Program DlUmn;
Const NMax = 2000;
Type Digit = 0..9; DlChislo = Array[1..Nmax] Of Digit;
Var S : String;
    M, N, R, F : DlChislo;
    I, MaxF : Word;
    Logic : Boolean;
{Процедура обнуления длинного числа}
Procedure Zero(Var A : DlChislo);
Var I : Integer;
  Begin
    For I := 1 To NMax Do A[I] := 0;
  End;
{Функция определения количества цифр в записи длинного числа}
Function Dlina(C : DlChislo) : Integer;
Var I : Integer;
 Begin
   I := NMax;
   While (I > 1) And (C[I] = 0) Do I := I - 1;
   Dlina := I
 End;
{Процедура печати длинного числа}
Procedure Print(A : DlChislo);
Var I : Integer;
 Begin
    For I := Dlina(A) DownTo 1 Do Write(A[I] : 1);
    WriteLn
 End;
{Процедура преобразования длинного числа в массив цифр}
Procedure Translate(S : String; Var A : DlChislo;
                    Var OK : Boolean);
Var I : Word;
 Begin
   Zero(A); I := Length(S); OK := True;
   While (I >= 1) And OK Do
   Begin
      If S[I] In ['0'..'9']
      Then A[Length(S) - I+ 1] := Ord(S[I]) - 48
      Else OK := False;
      I := I - 1
   End
 End;
Procedure Multiplication(A, B : DlChislo; Var C : DlChislo);
Var I, J : Integer; P : Digit; VspRez : 0..99;
```

```
 Begin
  Zero(C);
  For I := 1 To Dlina(A) Do
  Begin P := 0;
        For J := 1 To Dlina(B) Do
        Begin
          VspRez := A[I] * B[J] + P + C[I + J - 1];
     C[I + J - 1] := VspRez Mod 10;
     P := VspRez Div 10
   End;
        C[I + J] := P
  End
 End;
{Основная программа}
Begin
  Repeat {повторяем ввод, пока число не будет введено правильно}
    Write('Введите первый множитель: ');
    ReadLn(S); Translate(S, M, Logic)
  Until Logic;
  Repeat
    Write('Введите второй множитель: ');
    ReadLn(S); Translate(S, N, Logic)
  Until Logic;
  Multiplication(M, N, R); Print(R)
End.
```

В приведенном листинге **Print** — процедура вывода длинного числа. Предоставим читателю самостоятельно разобраться в алгоритме ее работы.

Вернемся к вычислению факториала. Используя разработанные подпрограммы, определим, значение факториала какого максимального числа можно разместить в памяти при таком представлении длинных чисел.

Вот измененный фрагмент основной программы, решающий поставленную задачу.

```
  Begin
    MaxF := 810;
    Zero(F);
    F[1] := 1;
    For I := 1 To MaxF Do
     Begin
       Str(I, S); {преобразование числа I к строковому типу S}
       Translate(S, M, Logic);
       Multiplication(F, M, F);
       Print(F);
       WriteLn('Факториал числа ', I : 4, ' содержит ', Dlina(F),
' цифр.')
     End
  End.
```

Расчеты показали, что можно вычислять факториалы до значения 810! включительно, в записи которого 1999 цифр. Далее вновь возникает переполнение.

Задание:

1. Составить программу сравнения двух многозначных чисел (количество знаков в записи чисел более 20).
2. Составить программу, суммирующую два натуральных многозначных числа с количеством знаков более 20.
3. Составить программу вычисления степени a^n, если a > MaxInt, n > 10.
4. Составить программу вычисления числа $2^{64} - 1$, в результате сохранить все цифры.
5. Составить программу вычисления 100!.
6. Составить программу извлечения точного квадратного корня из n-разрядного числа (n > 40).
7. Составить программу вычисления точного значения n!, где n > 12.

8. Составить программу вычисления точного значения n^n, где $n > 10$.
9. Составить программу деления числа a на число b, если a, b — многозначные числа.
10. Вычислить $100! + 2^{100}$.
11. Вычислить $100! – 2^{100}$.
12. Вычислить 7^{123}.
13. Встречаются ли среди цифр числа $2^{11213} – 1$ две подряд идущие девятки?
14. Вычислить $2^{–200}$.
15. Составить программу нахождения частного и остатка от деления m-значного числа на n–значное ($m, n > 20$).
16. Выяснить, какое из чисел a^m, b^n больше и на сколько ($a, b <= 40000$; $m, n <= 10$).
17. Найти n знаков в десятичной записи квадратного корня из целого числа m ($n >= 50$).
18. Найти количество делителей n-значного натурального числа ($n > 20$).
19. Вычислить точное значение $(n!)!$ ($n >= 3$).
20. Составить программу вычисления точного значения суммы $1! + 2! + 3! + ... + n!$ при $n > 10$.
21. Составить программу вычисления точного значения суммы дробей

$$\frac{1}{1!}+\frac{1}{2!}+\frac{1}{3!}+\cdots+\frac{1}{n!}$$

при $n > 10$. Ответ должен быть представлен в виде несократимой дроби P / Q, где P, Q — натуральные числа.
22. Вычислить точное значение $(n^n)!$ при $n >= 3$.
23. Составить программу вычисления точного значения суммы первых n членов последовательности $1, k, k^2, k^3, ..., k^n$ ($n >$ MaxInt). Указание: используйте формулу суммы n членов геометрической прогрессии.
24. Составить программу вычисления точного значения суммы первых n членов последовательности чисел, кратных данному натуральному числу k ($n >$ MaxInt). Указание: используйте формулу суммы n членов арифметической прогрессии.
25. Вычислить точное значение суммы $1^2 + 2^2 + 3^2 + … + n^2$ ($n >= 20000$).
26. Вычислить точное значение суммы $1^n + 2^n + 3^n + ... + n^n$ ($n >= 10$).
27. Найти первое простое число, которое больше 10^{11}.
28. Составить программу вычисления точного значения многочлена $a_nx^n + a_n - 1x^n - 1 + ... + a_1x + a_0$, где a_i и x — целые числа больше 10^{11}.
29. Найти наибольший общий делитель и наименьшее общее кратное чисел m и n ($m, n >= 10^{11}$).
30. Проверить, являются ли числа m и n ($m, n >= 10^{11}$) взаимно простыми.

Содержание отчета

– Название работы;
– Цель работы;
– Формулировку задания;
– Решенные задания, с пояснением решения в виде таблиц, рассуждений, рисунков.

Контрольные вопросы

1. В чем состоит алгоритм ввода, вывода длинного числа?
2. В чем состоят алгоритм сложения длинных чисел?
3. В чем состоит алгоритм умножения длинных чисел?

Список литературы:

Основная литература

1. Окулов С. М. Программирование в алгоритмах. / М. Окулов. – М.: БИНОМ. Лаборатория знаний, 2002. – 341 с.
2. Иванов Б. Н. Дискретная математика. Алгоритмы и программы. – М.: Лаборатория Базовых Знаний, 2002.

Дополнительная литература

1. Семакин И. Г. Основы программирования: учебник для сред. проф. образования / И. Г. Семакин, А. П. Шестаков. – М.: Издательский центр «Академия», 2007.

2. Акимов О.Е. Дискретная математика: логика, группы, графы. – М.: Лаборатория Базовых Знаний, 2001.
http://comp-science.narod.ru/DL-AR/dl-ar.html

Практическая работа № 12

Тема: Реализация классических задач комбинаторики.
Цель: закрепление умений по программированию алгоритмов.
Пояснения к работе:

В задачах, которые мы сейчас рассмотрим, элементы делятся на группы, и надо найти все способы такого раздела. При этом могут встретиться различные случаи. Иногда существенную роль играет порядок элементов в группах: например, когда сигнальщик вывешивает сигнальные флаги на нескольких мачтах, то для него важно не только то, на какой мачте окажется тот или иной флаг, но и то, в каком порядке эти флаги развешиваются. В других же случаях порядок элементов в группах никакой роли не играет. Когда игрок в домино выбирает кости из кучи, ему безразлично, в каком порядке они придут, а важен лишь окончательный результат.

Отличаются задачи и по тому, играет ли роль порядок самих групп. При игре в домино игроки сидят в определенном порядке, и важно не только то, как разделились кости, но и то, кому какие кости достались. Если раскладывать фотографии по одинаковым конвертам, чтобы разослать их, то существенно, как распределяются фотографии по конвертам, но порядок самих конвертов совершенно несущественен.

Играет роль и то, различаем ли мы между собой сами элементы или нет, а также различаем ли между собой группы, на которые делятся элементы. Наконец, в одних задачах некоторые группы могут оказаться пустыми, то есть не содержащими ни одного элемента, а в других такие группы недопустимы. В соответствии со всем сказанным возникает целый ряд различных комбинаторных задач на разбиение.

Задачи

Общая постановка этих задач:

Задача 1. Раскладка по ящикам

Даны n различных предметов и k ящиков. Надо положить в первый ящик n_1 предметов, во второй - n_2 предметов,..., в k -й - n_k предметов, где $n_1 + n_2 + \ldots + n_k = n.$ Сколькими способами можно сделать такое распределение?

Число различных раскладок по ящикам равно

$$P(n_1, n_2, \ldots, n_k) = \frac{n!}{n_1! n_2! \ldots n_k!}.$$

Эту формулу можно получить при решении следующей, на первый взгляд, совсем непохожей задачи:

Задача 2. Перестановки с повторением.

Имеются предметы k различных типов. Сколько различных перестановок можно сделать из n_1 предметов первого типа, n_2 предметов второго типа, ..., n_k предметов k -го типа? Число элементов в каждой перестановке равно $n_1 + n_2 + \ldots + n_k = n$. Поэтому если бы все элементы были различны, то число перестановок равнялось бы n ! . Но из-за того, что некоторые элементы совпадают, получится меньшее число перестановок. В самом деле, возьмем, например, перестановку

$$\underbrace{aa..a}_{n_1}\underbrace{bb\ldots b}_{n_2}\ldots\underbrace{xx\ldots x}_{n_3}, \qquad 5.1)$$

в которой сначала выписаны все элементы первого типа, потом все элементы второго типа, ..., наконец, все элементы k -го типа. Элементы первого типа можно переставлять друг с другом n_1 ! способами. Но так как все эти элементы одинаковы, то такие перестановки ничего не меняют. Точно так же ничего не меняют n_2 ! перестановок элементов второго типа, ..., n_k ! перестановок элементов k -го типа.

Перестановки элементов первого типа, второго типа и так далее можно делать независимо друг от друга. Поэтому элементы перестановки 5.1. можно переставлять друг с другом $n_1! n_2! \ldots n_k$! способами так, что она остается неизменной. То же самое верно и для любого другого расположения элементов. Поэтому множество всех n ! перестановок распадается

на части, состоящие из $n_1!n_2!\ldots n_k!$ одинаковых перестановок каждая. Значит, число различных перестановок с повторениями, которые можно сделать из данных элементов, равно

$$P(n_1, n_2, \ldots, n_k) = \frac{n!}{n_1!n_2!\ldots n_k!}, \quad 5.2)$$

где $n_1 + n_2 + \ldots + n_k = n.$

Пользуясь формулой 5.2, можно ответить на вопрос: сколько перестановок можно сделать из букв слова "Миссисипи"? Здесь у нас одна буква "м", четыре буквы "и", три буквы "с" и одна буква "п", а всего 9 букв. Значит, по формуле 5.2 число перестановок равно

$$P(4, 3, 1, 1) = \frac{9!}{4! \cdot 3! \cdot 1! \cdot 1!} = 2520.$$

Чтобы установить связь между этими задачами, занумеруем все n мест, которые могут занимать наши предметы. Каждой перестановке соответствует распределение номеров мест на k классов. В первый класс попадают номера тех мест, на которые попали предметы первого типа, во второй - номера мест предметов второго типа и так далее. Тем самым устанавливается соответствие между перестановками с повторениями и раскладкой номеров мест по "ящикам". Понятно, что формулы решения задач оказались одинаковыми.

В рассмотренных задачах мы не учитывали порядок, в котором расположены элементы каждой части. В некоторых задачах этот порядок надо учитывать.

Задача 3. Флаги на мачтах.

Имеется n различных сигнальных флагов и k мачт, на которые их вывешивают. Значение сигнала зависит от того, в каком порядке развешены флаги. Сколькими способами можно развесить флаги, если все флаги должны быть использованы, но некоторые из мачт могут оказаться пустыми?

Каждый способ развешивания флагов можно осуществить в два этапа. На первом этапе мы переставляем всеми возможными способами данные n флагов. Это можно сделать n ! способами. Затем берем один из способов распределения n одинаковых флагов по k мачтам (число этих способов C_{n+k-1}^{k-1}). Пусть этот способ заключается в том, что на первую мачту надо повесить n_1 флагов, на вторую - n_2 флагов, ..., на k -ю n_k флагов, где $n_1 + n_2 + \ldots + n_k = n.$ Тогда берем первые n_1 флагов данной последовательности и развешиваем в полученном порядке на первой мачте; следующие n_2 флагов развешиваем на второй мачте и т.д. Ясно, что используя все перестановки n флагов и все способы распределения n одинаковых флагов по k мачтам, получим все способы решения поставленной задачи. По правилу произведения получаем, что число способов развешивания флагов равно

$$n!C_{n+k-1}^{k-1} = \frac{(n+k-1)!}{(k-1)!} = A_{n+k-1}^{n}. \quad 5.3)$$

Вообще, если имеется n различных вещей, то число способов распределения этих вещей по k различным ящикам равно A_{n+k-1}^{n}.

Разные статистики

Задачи о раскладке предметов по ящикам весьма важны для статистической физики. Эта наука изучает, как распределяются по своим свойствам физические частицы; например, какая часть молекул данного газа имеет при данной температуре ту или иную скорость. При этом множество всех возможных состояний распределяют на большое число k маленьких ячеек (фазовых состояний), так что каждая из n частиц попадет в одну из ячеек.

Вопрос о том, какой статистике подчиняются те или иные частицы, зависит от вида этих частиц. В классической статистической физике, созданной Максвеллом и Больцманом, частицы считаются различимыми друг от друга. Такой статистике подчиняются, например, молекулы газа. Известно, что n различных частиц можно распределить по k ячейкам k^n способами. Если все эти k^n способов при заданной энергии имеют равную вероятность, то говорят о статистике Максвелла-Больцмана.

Оказалось, что этой статистике подчиняются не все физические объекты. Фотоны, атомные ядра и атомы, содержащие четное число элементарных частиц, подчиняются иной статистике, разработанной Эйнштейном и индийским ученым Бозе. В статистике Бозе-Эйнштейна частицы считаются неразличимыми друг от друга. Поэтому имеет значение лишь то, сколько частиц попало в ту или иную ячейку, а не то, какие именно частицы туда попали.

Однако для многих частиц, например таких как электроны, протоны и нейтроны, не годится и статистика Бозе-Эйнштейна. Для них в каждой ячейке может находится не более одной частицы, причем различные распределения, удовлетворяющие указанному условию, имеют равную вероятность. В этом случае может быть C_k^n различных распределений. Эта статистика называется статистикой Дирака-Ферми.

Деревья и перестановки из n элементов

С помощью леса можно представить перестановки из n элементов множества $M = \{a; b; c; d\}$ (множество мы определяем так: множество - это неупорядоченная совокупность различных объектов или структура данных, используемая для представления множества). Подсчитаем, сколько можно получить перестановок. Для n такой лес изображен на рис. 5.1.

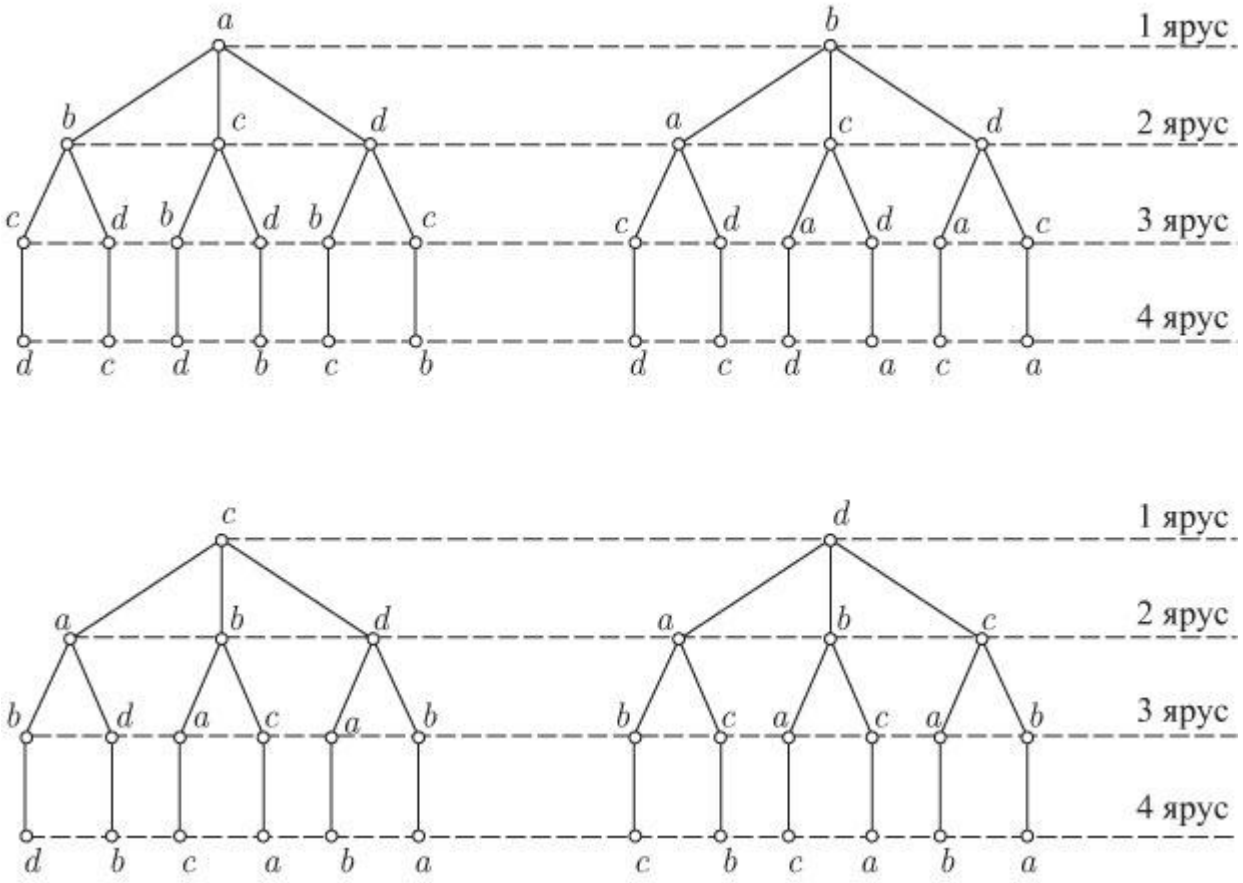


Рис. 5.1. Всевозможные перестановки прочитываются по этой схеме от корневой до висячей вершины соответствующего дерева. Ярус показывает номер места, на котором расположен элемент. Число висячих вершин леса равно числу перестановок

Число сочетаний

Рассмотрим подмножества множества, состоящего из пяти элементов, и подсчитаем их число. При этом записывать подмножества будем не с помощью букв, как обычно, а в виде последовательностей длиной пять, составленных из нулей и единиц. Каждая из единиц указывает на наличие в подмножестве соответствующего элемента. Например, подмножества, содержащие один элемент, будут изображаться следующими последовательностями: 10000, 01000, 00100, 00010, 00001. Пустое подмножество $\emptyset$ будет соответствовать последовательности 00000. Подмножества, содержащие по два элемента из пяти, запишутся с помощью следующих последовательностей: 11000, 10100, 10010, 10001, 01100. 01010, 01001, 00110, 00101, 00011. Всего их $C_5^2 = 10.$

Вообще, число сочетаний из n элементов по m равно числу всевозможных последовательностей из m единиц и $n - m$ нулей.

Задачи на разбиение чисел

Теперь мы переходим к задачам, в которых все разделяемые предметы совершенно

одинаковы. В этом случае можно говорить не о разделении предметов, а о разбиении натуральных чисел на слагаемые (которые, конечно, тоже должны быть натуральными числами).

Здесь возникает много различных задач. В одних задачах учитывается порядок слагаемых, в других - нет.

Задача 4. Отправка бандероли.

За пересылку бандероли надо уплатить 18 рублей. Сколькими способами можно оплатить ее марками стоимостью 4, 6, и 10 рублей, если два способа, отличающиеся порядком марок, считаются различными (запас марок различного достоинства считаем неограниченным)?

Обозначим через $f(N)$ число способов, которыми можно наклеить марки в 4, 6 и 10 рублей так, чтобы общая стоимость этих марок равнялась $N.$ Тогда для $f(N)$ справедливо следующее соотношение:

$$f(N) = f(N-4) + f(N-6) + f(N-10). \quad 5.4)$$

Пусть имеется некоторый способ наклейки марок с общей стоимостью $N,$ и пусть последней наклеена марка стоимостью 4 рубля. Тогда все остальные марки стоят ($N-4$) рубля. Наоборот, присоединяя к любой комбинации марок общей стоимостью ($N-4$) рубля одну четырехрублевую марку, получаем комбинацию марок стоимостью N рублей. При этом из разных комбинаций стоимостью ($N-4$) рублей получается разные комбинации стоимостью N рублей. Итак, число искомых комбинаций, где последней наклеена марка стоимостью 4 рубля, равно $f(N-4).$

Точно так же доказывается, что число комбинаций, оканчивающихся на шестирублевую марку, равно $f(N-6),$ а на десятирублевую марку оканчиваются $f(N-10)$ комбинацией. Поскольку любая комбинация оканчивается на марку одного из указанных типов, то по правилу суммы получаем соотношение 5.4.

Соотношение 5.4 позволяет свести задачу о наклеивании марок на сумму N рублей к задачам о наклеивании марок на меньшие суммы. Но при малых значениях N задачу легко решить непосредственно. Простой подсчет показывает, что

$$f(0) = 1, f(1) = f(2) = f(3) = 0, f(4) = 1, f(5) = 0,$$
$$f(6) = 1, f(7) = 0, f(8) = 1, f(9) = 0.$$

Равенство $f(0) = 1$ означает, что сумму в 0 рублей можно уплатить единственным образом: совсем не наклеивая марок. А сумму в 1,2,3,5,7 и 9 рублей вообще никак нельзя получить с помощью марок стоимостью 4, 6 и 10 рублей. Используя значения $f(N)$ для $N = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,$ легко найти $f(10):$

$$f(10) = f(6) + f(4) + f(0) = 3.$$

После этого находим

$$f(11) = f(7) + f(5) + f(1) = 0,$$
$$f(12) = f(8) + f(6) + f(2) = 2$$

и т.д. Наконец, получаем значение $f(18) = 8.$ Таким образом, марки можно наклеить восемью способами. Эти способы таковы: $10, 4, 4$; $4, 10, 4$; $4, 4, 10$; $6, 4, 4, 4$; $4, 6, 4, 4$; $4, 4, 6, 4$; $4, 4, 4, 6$; $6, 6, 6.$ Отметим, что значения $f(N)$ для $N = 1, 2, 3, 4, 5, 6, 7, 8, 9$ можно было получить иначе, не приводя непосредственно проверки. Дело в том, что при $N < 0$ имеем $f(N) = 0,$ поскольку отрицательную сумму нельзя уплатить, наклеивая неотрицательное количество марок. В то же время, как мы видели, $f(0) = 1.$ Поэтому $f(1) = f(-3) + f(-5) + f(-9) = 0.$

Точно так же получаем значение $f(2) = 0, f(3) = 0,$ а для $N = 4$ имеем $f(4) = f(0) + f(-2) + f(-6) = 1.$

Задача 5.Общая задача о наклейке марок.

Разобранная задача является частным случаем следующей общей задачи: Имеются марки

достоинством в $n_1, n_2, \ldots, n_k$. Сколькими способами можно оплатить с их помощью сумму в N рублей, если два способа, отличающиеся порядком, считаются различными? Все числа $n_1, n_2, \ldots, n_k$ различны, а запас марок неограничен. Здесь на первом месте мы будем указывать число слагаемых, на втором – разбиваемое число и на последнем - ограничения на величину слагаемых.

В этом случае число $f(N)$ способов удовлетворяет соотношению

$$f(N) = f(N - n_1) + f(N - n_2) + \ldots + f(N - n_k). \quad 5.5)$$

При этом $f(N) = 0,$ если $f(0) = 1$ и $N < 0.$ С помощью соотношения 5.5 можно найти $f(N)$ для любого $N,$ последовательно вычисляя $f(1), f(2), \ldots, f(N-1).$

Рассмотрим частный случай этой задачи, когда $n_1 = 1, n_2 = 2, \ldots, n_k = k.$ Мы получаем всевозможные разбиения числа N на слагаемые $1, 2, \ldots, k,$ причем разбиения, отличающиеся порядком слагаемых, считаются различными. Обозначим число этих разбиений через $\varphi(k; N).$ (На первом месте мы будем указывать число слагаемых, на втором - разбиваемое число и на последнем – ограничения на величину слагаемых.) Из соотношения 5.5 следует, что

$$\varphi(k; N-1) + \varphi(k; N-2) + \ldots + \varphi(k; N-k). \quad 5.6)$$

При этом $\varphi(k; 0) = 1$ и $\varphi(k; N) = 0,$ если $N < 0.$ Вычисление $\varphi(N; k)$ можно упростить, если заметить, что

$$\varphi(N-1; k) = \varphi(N-2; k) + \varphi(N-k-1; k),$$

и потому

$$\varphi(N; k) = 2\varphi(N-1; k) - \varphi(N-k-1; k). \quad 5.7)$$

Ясно, что слагаемые не могут быть больше $N.$ Поэтому $\varphi(N, N)$ равно числу всех разбиений на N на натуральные слагаемые (включая и "разбиение" $N = N.$ Если число слагаемых равно $s,$ то получаем C_{N-1}^{s-1} разбиений. Поэтому

$$\varphi(N, N) = C_{N-1}^{0} + C_{N-1}^{1} + \ldots + C_{N-1}^{N-1} = 2^{N-1}.$$

Итак, мы доказали, что натуральное число N можно разбить на слагаемые 2^{N-1} способами. Напомним, что при этом учитывается порядок слагаемых.Например, число 5 можно разбить на слагаемые $2^{5-1} = 16$ способами.

5 = 5	5 = 3 + 1 + 1	5 = 1 + 2 + 2
5 = 4 + 1	5 = 1 + 3+ 1	5 = 2 + 1 + 1 + 1
5 = 1 + 4	5 = 1 + 1 + 3	5 = 1 + 2 + 1 + 1
5 = 2 + 3	5 = 2 + 2 + 1	5 = 1 + 1 + 2 + 1
5 = 3 + 2	5 = 2 + 1 + 2	5 = 1 + 1 + 1 + 2
		5 = 1 + 1 + 1 + 1 + 1

Комбинаторные задачи теории информации

Информация - сведения, неизвестные до их получения, или данные, или значения, приписанные данным.

Теория информации - математическая дисциплина, изучающая количественные свойства информации.

Задачу, похожую на только что решенную, приходится решать в теории информации. Предположим, что сообщение передается с помощью сигналов нескольких типов. Длительность передачи сигнала первого типа равна $t_1,$ второго типа - $t_2, \ldots, k$ -го типа - t_k единиц времени.

Задача 6. Сколько различных сообщений можно передать с помощью этих сигналов за T единиц времени? При этом учитываются лишь "максимальные" сообщения, то есть сообщения, к которым нельзя присоединить ни одного сигнала, не выйдя за рамки отведенного для передачи времени.

Обозначим число сообщений, которые можно передать за время T через $f(T)$. Рассуждая точно так же, как и в задаче о марках, получаем, что $f(T)$ удовлетворяет соотношению

$$f(T) = f(T - t_1) + \ldots + f(T - t_k). \quad (5.8)$$

При этом снова $f(T) = 0,$ если $T < 0$ и $f(0) = 1$.

Задание:

1. Запрограммируйте предложенные алгоритмы.

Содержание отчета

– Название работы;
– Цель работы;
– Формулировку задания;
– Решенные задания, с пояснением решения в виде таблиц, рассуждений, рисунков.

Контрольные вопросы

1. Перечислите основные законы комбинаторики.
2. Приведите примеры на применение знаний по комбинаторике.

Список литературы:

Основная литература

1. Окулов С. М. Программирование в алгоритмах. / М. Окулов. – М.: БИНОМ. Лаборатория знаний, 2002. – 341 с.
2. Иванов Б. Н. Дискретная математика. Алгоритмы и программы. – М.: Лаборатория Базовых Знаний, 2002.

Дополнительная литература

1. Семакин И. Г. Основы программирования: учебник для сред. проф. образования / И. Г. Семакин, А. П. Шестаков. – М.: Издательский центр «Академия», 2007.
2. Акимов О.Е. Дискретная математика: логика, группы, графы. – М.: Лаборатория Базовых Знаний, 2001.

Интернет ресурсы

1. Костюкова Нина Ивановна Комбинаторные алгоритмы для программистов http://www.intuit.ru/studies/courses/65/65/info

Практическая работа № 13

Тема: Реализация генерации комбинаторных объектов.

Цель: закрепление умений по программированию алгоритмов.

Пояснения к работе:

Множества и мультимножества

Не существует формального определения множества; считается что это понятие первичное и не определяется. Так, можно говорить, что множество есть объединение различных элементов, но при этом мы оставляем неопределяемыми понятия "объединение" и "элементы". Дадим следующее определение множеству: множество - это неупорядоченная совокупность различных объектов или структура данных, используемая для представления множества. Мультимножество есть объединение не обязательно различных элементов; его можно считать множеством, в котором каждому элементу поставлено в соответствие положительное целое число, называемое кратностью.

Конечное множество S будем записывать в следующем виде:

$S = \{s_1, s_2, \ldots, s_n\},$

где $s_1, s_2, \ldots, s_n$ - элементы S, обязательно различные! Мощность множества S

обозначается как $|S|$, для выписанного выше множества мощность записывается так $|S| = n$. Если S - конечное мультимножество, то будем записывать его в следующем виде:

$$S = \{\underbrace{s_1, s_1, \ldots, s_1}_{m_1 \text{раз}}, \underbrace{s_2, s_2, \ldots, s_2}_{m_2 \text{ раз}}, \underbrace{s_3, s_3, \ldots, s_3}_{m_3 \text{ раз}}\} = \{m_1 \bullet s_1, m_2 \bullet s_2, \ldots, m_n \bullet s_n\}.$$

Здесь все s_i различны и m_i - кратность элемента s_i. В этом случае мощность S равна

$$|S| = \sum_{i=1}^{n} m_i.$$

Наиболее общими операциями на множествах и мультимножествах являются операции объединения и пересечения. Для множеств эти операции будем обозначать $\cup$ и $\cap$, а для мультимножеств - $\overset{+}{\cup}$ и $\overset{+}{\cap}$. Последовательное и связанное представление последовательностей можно использовать для множеств и мультимножеств очевидным способом. Индуцируя искусственный порядок элементов множества или используя собственный порядок, если он существует, можно рассматривать множество как последовательность. Аналогично, как последовательность можно рассматривать и мультимножество, или, для того чтобы сэкономить место, его можно рассматривать как последовательность пар, каждая из которых состоит из элемента и его кратности.

Как и для последовательностей, наилучший метод представления множеств или мультимножеств существенно зависит от операций, которые выполняются над ними. Предположим, например, что имеем дело с непересекающимися подмножествами множества $S = \{s_1, s_2, \ldots, s_n\}$ и что над ними необходимо выполнить две следующие операции: объединение двух множеств и отыскание подмножества, содержащего данное s_i. Таким образом, в любой момент времени имеем разбиение S на непустые непересекающиеся подмножества. Рассмотрим эти операции в конце данной лекции.

С целью идентификации считаем, что каждое из непересекающихся подмножеств множества S имеет имя. Имя - это просто один из элементов подмножества, или, иначе, - представитель подмножества. Когда мы будем ссылаться на имя подмножества, то будем под этим подразумевать его представителя. Рассмотрим, например, множество

$$S = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\},$$

разбитое на четыре непересекающихся подмножества

$$\{1, 6, \langle 7\rangle, 8, 11\}\{\langle 2\rangle\}\{\langle 3\rangle, 4, 5\}\{9, \langle 10\rangle\} \quad 6.1)$$

В каждом из подмножеств, взятый в скобки элемент является его именем. Если нам нужно найти подмножество, в котором содержится восьмерка, искомым ответом будет 7, то есть имя подмножества, содержащего восьмерку. Если нужно взять объединение подмножеств с именами 2 и 10, получим разбиение множества S следующего вида:

$$\{1, 6, \langle 7\rangle, 8, 11\}\{\langle 3\rangle, 4, 5\}\{\langle 2\rangle\} \cup \{9, \langle 10\rangle\}$$

Именем множества $\{\langle 2\rangle\} \cup \{9, \langle 10\rangle\}$ может быть или 2, или 10.Предполагаем, что вначале имеется разбиение множества $S = \{s_1, s_2, \ldots, s_n\}$ на n подмножеств, каждое из которых состоит из одного элемента

$$\{\langle s_1\rangle\}\{\langle s_2\rangle\}, \ldots, \{\langle s_n\rangle\} \quad 6.2)$$

и имя каждого из них есть просто этот единственный элемент. Это разбиение преобразуется путем применения операций объединения вперемешку с операциями отыскания. Такая кажущаяся на первый взгляд надуманной задача чрезвычайно полезна в определенных комбинаторных алгоритмах; пример ее полезности виден в "жадном" алгоритме (лекция 16).

Для реализации операций и объединения, и отыскания опишем процедуры (операции) $UNION(x, y)$ и $FIND(x)$. Процедура (операция) $UNION(x, y)$ по именам двух различных подмножеств x и y образует новое подмножество, содержащее все элементы множеств x и y. Процедура (операция) $FIND(x)$ выдает имя множества, содержащего x. Например, если нужно множество,содержащее a, объединить с множеством, содержащим b, необходимо выполнить следующую последовательность операторов:

$x \leftarrow FIND(a)$
$y \leftarrow FIND(b)$
$if\, x \neq y\ then\, UNION(x, y).$

Предположим, что мы имеем u операций объединения, перемешанных с f операциями отыскания, и что начинаем алгоритм с множества $S = \{s_1, s_2, \ldots, s_n\}$, которое разбито на подмножества, состоящие из одного элемента (см. 6.2.). Найдем такую структуру данных для представления непересекающихся подмножеств множества S, чтобы последовательность операций можно было производить эффектно. Такой структурой данных является представление в виде леса с указателями отца, как показано на рис. 4.5 лекции 4. Каждый элемент s_i множества будет узлом леса, а отцом его будет элемент из того же подмножества, что и s_i. Если элемент не имеет отца, то есть является корнем, то он будет именем своего подмножества. В соответствии с этим разбиение 6.1 может быть представлено так:

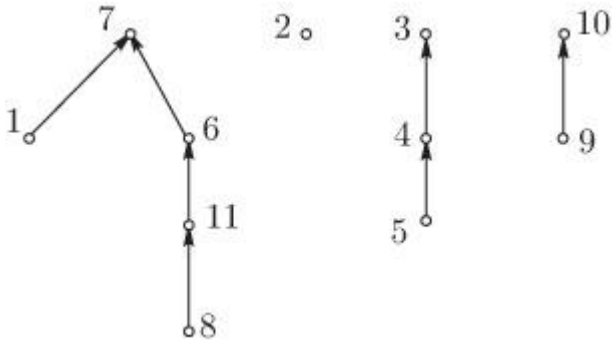
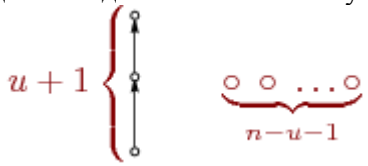


Рис. 6.1. Представление разбиения

При таком представлении процедура (операция) $FIND(x)$ состоит в переходах по указателям отцов от x до корня, то есть имени, его подмножества. Процедура (операция) $UNION(x, y)$ состоит в связывании вместе некоторым образом деревьев, имеющих корни x и y. Например, такую связь можно осуществить, сделав y отцом x.

После u операций объединения наибольшее из возможных подмножеств, получающихся в результате разбиения S, будет содержать $u + 1$ элементов. Поскольку каждое объединение уменьшает число подмножеств на единицу, последовательность операций может содержать не более $n - 1$ объединений, откуда $u \leqslant n - 1$. Так как каждая операция объединения изменяет имя подмножества, содержащего некоторые элементы, можно считать, что каждому объединению предшествует по крайней мере одно отыскание, в связи с чем естественно предположить, что $f \geqslant u$. Выясним, насколько эффективно можно выполнить последовательность из $u \leqslant n - 1$ операций объединения, перемешанных с $f \geqslant u$ операциями отыскания. Время, требуемое на операции объединения, очевидно, пропорционально u, потому что необходимая для каждой операции объединения переделка некоторых указателей требует фиксированного количества работы. Поэтому сосредоточим свое внимание на времени, требуемом для f операций отыскания.

Если операция $UNION(x, y)$ выполняется путем назначения x отцом y, то после u операций объединения может получиться лес, показанный ниже.



В этом случае, если f операций отыскания выполняются после всех операций объединения и каждый поиск начинается внизу цепи из $u + 1$ элементов множества, то ясно, что время, требуемое на операции отыскания, будет пропорционально $f \cdot (u + 1)$. Очевидно, оно не может быть больше, чем константа, умноженная на $f \cdot (u + 1)$. Можно существенно уменьшить

эту оценку.

Формула включений и исключений

Пусть имеется N предметов, некоторые из которых обладают свойствами $\alpha_1, \alpha_2, \ldots \alpha_n$. При этом каждый предмет может либо не обладать ни одним из этих свойств, либо обладать одним или несколькими свойствами. Обозначим через $N(\alpha_i \alpha_j \ldots \alpha_k)$ количество предметов, обладающих свойствами $\alpha_i, \alpha_j, \ldots, \alpha_k$ (и быть может, еще некоторыми из других свойств). Если нужно взять предметы, не обладающие некоторым свойством, то эти свойства пишем со штрихом. Например, через $N(\alpha_1 \alpha_2 \alpha_4'$) обозначено количество предметов, обладающих свойствами α_1, α_2, но не обладающих свойством α_4 (вопрос об остальных свойствах останется открытым). Число предметов, не обладающих ни одним из указанных свойств, обозначается по этому правилу через $N(\alpha_1' \alpha_2' \ldots \alpha_n')$. Общий закон состоит в том, что

$$N(\alpha_1' \alpha_2' \ldots \alpha_n') = N - N(\alpha_1) - N(\alpha_2) - \ldots - N(\alpha_n) + N(\alpha_1 \alpha_2) + \\ + N(\alpha_1 \alpha_3) + \ldots + N(\alpha_1 \alpha_n) + \ldots + N(\alpha_{n-1} \alpha_n) - N(\alpha_1 \alpha_2 \alpha_3) - \\ - N(\alpha_{n-2} \alpha_{n-1} \alpha_n + \ldots + (-1)^n N(\alpha_1 \alpha_2 \ldots \alpha_n). \qquad 6.3)$$

Здесь алгебраическая сумма распространена на все комбинации свойств $\alpha_1, \alpha_2, \ldots, \alpha_n$ (без учета их порядка), причем знак + ставится, если число учитываемых свойств четно, и знак $-$, если это число нечетно. Например, $N(\alpha_1 \alpha_3 \alpha_6 \alpha_8)$ входит со знаком +, а $N(\alpha_3 \alpha_4 \alpha_{10})$ со знаком $-$. Формулу 6.3 называют формулой включений и исключений - сначала исключаются все предметы, обладающие хотя бы одним из свойств $\alpha_1, \alpha_2, \ldots, \alpha_n$, потом включаются предметы, обладающие, по крайней мере, двумя из этих свойств, затем исключаются имеющие, по крайней мере, три и т.д.

Решето Эратосфена

Одной из самых больших загадок математики является расположение простых чисел в ряду всех натуральных чисел. Иногда два простых числа идут через одно, (например, 17 и 19, 29 и 31), а иногда подряд идет миллион составных чисел. Сейчас ученые знают уже довольно много о том, сколько простых чисел содержится среди N первых натуральных чисел. В этих подсчетах весьма полезным оказался метод, восходящий еще к древнегреческому ученому Эратосфену. Он жил в третьем веке до новой эры в Александрии.

Эратосфен занимался самыми различными вопросами - ему принадлежат интересные исследования в области математики, астрономии и других наук. Впрочем, такая разносторонность привела его к некоторой поверхностности. Современники несколько иронически называли Эратосфена "во всем второй": второй математик после Евклида, второй астроном после Гиппарха и т.д.

В математике Эратосфена интересовал как раз вопрос о том, как найти все простые числа среди натуральных чисел от 1 до N. (Эратосфен считал 1 простым числом. Сейчас математики считают 1 числом особого вида, которое не относится ни к простым, ни к составным числам.) Он придумал для этого следующий способ. Сначала вычеркивают все числа, делящиеся на 2 (исключая само число 2). Потом берут первое из оставшихся чисел (а именно 3). Ясно, что это число - простое. Вычеркивают все идущие после него числа, делящиеся на 3. Первым оставшимся числом будет 5. Вычеркивают все идущие после него числа, делящиеся на 5, и т.д. Числа, которые уцелеют после всех вычеркиваний, и являются простыми. Так как во времена Эратосфена писали на восковых табличках и не вычеркивали, а "выкалывали" цифры, то табличка после описанного процесса напоминала решето. Поэтому метод Эратосфена для нахождения простых чисел получил название "решето Эратосфена".

Подсчитаем, сколько останется чисел в первой сотне, если мы вычеркнем по методу Эратосфена числа, делящиеся на 2, 3 и 5. Иными словами, поставим такой вопрос: сколько чисел в первой сотне не делится ни на одно из чисел 2, 3, 5? Эта задача решается по формуле включения и исключения.

Обозначим через α_1 свойство числа делиться на 2, через α_2 - свойство делимости на 3 и через α_3 - свойство делимости на 5. Тогда $\alpha_1 \alpha_2$ означает, что число делится на 6, $\alpha_1 \alpha_3$ означает, что оно делится на 10, и $\alpha_2 \alpha_3$ - оно делится на 15. Наконец, $\alpha_1 \alpha_2 \alpha_3$ означает, что число делится на 30. Надо найти, сколько чисел от 1 до 100 не делится ни на 2, ни на 3, ни на 5, то

есть не обладает ни одним из свойств α_1, α_2, α_3. По формуле 6.3 имеем

$$N(\alpha_1'\alpha_2'\alpha_3') = 100 - N(\alpha_1) - N(\alpha_2) - N(\alpha_3) + N(\alpha_1\alpha_2) +$$
$$+ N(\alpha_1\alpha_3) + N(\alpha_2\alpha_3) - N(\alpha_1\alpha_2\alpha_3).$$

Но чтобы найти, сколько чисел от 1 до N делится на n, надо разделить N на n и взять целую часть получившегося частного. Поэтому

$$N(\alpha_1) = 50, N(\alpha_2) = 33, N(\alpha_3) = 20,$$
$$N(\alpha_1\alpha_2) = 16, N(\alpha_1\alpha_3) = 10, N(\alpha_2\alpha_3) = 6, N(\alpha_1\alpha\alpha_3) = 3,$$

и значит,

$$N(\alpha_1'\alpha_2'\alpha_3') = 32.$$

Таким образом, 26 числа от 1 до 100 не делятся ни на 2, ни на 3, ни на 5. Эти числа и уцелеют после первых трех шагов процесса Эратосфена. Кроме них останутся сами числа 2, 3 и 5. Всего останется 35 чисел.

А из первой тысячи после первых трех шагов процесса Эратосфена останется 335 чисел. Это следует из того, что в этом случае

$$N(\alpha_1) = 500, N(\alpha_2) = 333, N(\alpha_3) = 200,$$
$$N(\alpha_1\alpha_2) = 166, N(\alpha_1\alpha_3) = 100, N(\alpha_2\alpha_3) = 66, N(\alpha_1\alpha_2\alpha_3) = 33.$$

Примеры программы

Программа 1. Решето Эратосфена.

```
{В примере, иллюстрирующем работу с множествами,  реализуется алгоритм
выделения из первой сотни натуральных чисел всех простых чисел. В основе
алгоритма лежит прием "решета Эратосфена".
Алгоритм  написан на языке программирования  Turbo-Pascal.}

Uses crt;
Const
    N=100; {количество элементов исходного множества}
Type
    SetN=set of 1..N;
var
    n1, next, I : word; {вспомогательные переменные }
    BeginSet,              {исходное множество }
    PrimerSet: SetN;   {множество простых чисел }
Begin
    Clrscr;   {почистить экран}
    BeginSet:=[2..N]; {создать исходное множество}
    PrimerSet:=[1];     {первое простое число}
    next:=2;                {следующее простое число}
while BeginSet <> [ ] do  {начало основного цикла}
    begin
        n1:=next; {n1-число, кратное очередному простому (next)}
        while n1<=N do
        {цикл удаления из исходного множества непростых чисел}
        begin
            BeginSet:=BeginSet-[n1];
            n1:=n1+next {следующее кратное}
            end;               {конец цикла удаления}
repeat    {получить следующее простое число, которое есть первое
           не вычеркнутое из исходного множества}
inc(next)
until(next  in  BeginSet) or (next > N)
end;  {конец основного цикла}
{вывод результата}
textcolor(15);  {задание цвета}
for I:=1 to N do
    if i in PrimerSet then write(I:8);
    readln;
end.
```

Программа 2. Простые числа в порядке убывания от 200.

```
{Находит и пишет все простые числа в порядке убывания от 2 до 200.
Алгоритм  написан на языке программирования   Turbo-Pascal.}
```

```
Uses crt;
const
n=197;
var

i,q,w,e,r,t:integer;
prost:array[1..n] of integer;

begin
clrscr;
e:=1;
r:=0;
for q:=1 to n do begin
r:=0;
for w:=2 to n-1 do
if (q<>w) and (q mod w =  0) then r:=1;
{prost[e]:=q; e:=e+1;}

if r=0 then begin{begin write(q,' ');}

prost[e]:=q;
e:=e+1;
end;
end;

for i:=e downto 2 do  begin
write (prost[i],' ');
if wherex>70 then writeln;

end;
readln;
end.
```

Программа 3. Поиск литер в строке.

```
{Поиск числа вхождений в данную сроку литер a, c, e, h.
Алгоритм  написан на языке программирования  Turbo-Pascal.}

Uses crt;
type
liter_set = set of char;

var
c:integer;
let: liter_set;
a:char;

begin
 clrscr;
 let:=['a','c','e','h'];

repeat
  a:=readkey;
  write(a);
  if a in let then c:=c+1;
 until a = '.';
 writeln;
 writeln('Общее число вхожений литер a,c,e,h в вашу запись:',c);
readln;

end.
```

Размещения без повторений

Имеется n различных предметов. Сколько из них можно составить k -расстановок? При этом две расстановки считаются различными, если они либо отличаются друг от друга хотя бы одним элементом, либо состоят из одних и тех же элементов, но расположенных в разном порядке. Такие расстановки называют размещениями без повторений, а их число обозначают A_n^k. При

составлении k -размещений без повторений из n предметов нам надо сделать k выборов. На первом шагу можно выбрать любой из имеющихся n предметов. Если этот выбор уже сделан, то на втором шагу приходится выбирать из оставшихся $n-1$ предметов. На k - м шагу $n-k+1$ предметов. Поэтому по правилу произведения получаем, что число k - размещений без повторения из n предметов выражается следующим образом:

$$A_n^k = n(n-1)\ldots(n-k+1).$$

Перестановки

При составлении размещений без повторений из n элементов по k мы получили расстановки, отличающиеся друг от друга и составом, и порядком элементов. Но если брать расстановки, в которые входят все n элементов, то они могут отличаться друг от друга лишь порядком входящих в них элементов. Такие расстановки называют перестановками из n элементов, или, короче, n - перестановками.

Сочетания

В тех случаях, когда нас не интересует порядок элементов в комбинации, а интересует лишь ее состав, говорят о сочетаниях. Итак, k - сочетаниями из n элементов называют всевозможные k - расстановки, составленные из этих элементов и отличающиеся друг от друга составом, но не порядком элементов. Число k -сочетаний, которое можно составить из n элементов, обозначают через C_n^k.

Формула для числа сочетаний получается из формулы для числа размещений. В самом деле, составим сначала все k -сочетания из n элементов, а потом переставим входящие в каждое сочетание элементы всеми возможными способами. При этом получается, что все k -размещения из n элементов, причем каждое только по одному разу. Но из каждого k - сочетания можно сделать k ! перестановок, а число этих сочетаний равно C_n^k. Значит справедлива формула

$$k!C_n^k = A_n^k.$$

Из этой формулы находим, что

$$C_n^k = \frac{A_n^k}{k!} = \frac{n!}{(n-k)!k!}.$$

Рекуррентные соотношения

При решении многих комбинаторных задач пользуются методом сведения данной задачи к задаче, касающейся меньшего числа предметов. Метод сведения к аналогичной задаче для меньшего числа предметов называется методом рекуррентных соотношений (от латинского "recurrere" - "возвращаться").

Понятие рекуррентных соотношений проиллюстрируем классической проблемой, которая была поставлена около 1202 года Леонардо из Пизы, известным как Фибоначчи. Важность чисел Фибоначчи для анализа комбинаторных алгоритмов делает этот пример весьма подходящим.

Фибоначчи поставил задачу в форме рассказа о скорости роста популяции кроликов при следующих предположениях. Все начинается с одной пары кроликов. Каждая пара становится фертильной через месяц, после чего каждая пара рождает новую пару кроликов каждый месяц. Кролики никогда не умирают, и их воспроизводство никогда не прекращается.

Пусть F_n - число пар кроликов в популяции по прошествии n месяцев, и пусть эта популяция состоит из N_n пар приплода и O_n "старых" пар, то есть $F_n = N_n + O_n$. Таким образом, в очередном месяце произойдут следующие события: $O_{n+1} = O_n + N_n = F_n$. Старая популяция в $(n+1)$ -й момент увеличится на число родившихся в момент времени n. $N_{n+1} = O_n$. Каждая старая пара в момент времени n производит пару приплода в момент времени $(n+1)$. В последующий месяц эта картина повторяется:

$$O_{n+2} = O_{n+1} + N_{n+1} = F_{n+1},$$
$$N_{n+2} = O_{n+1}$$

Объединяя эти равенства, получим следующее рекуррентное соотношение:

$$F_{n+2} = O_{n+2} + N_{n+2} = F_{n+1} + O_{n+1},$$

$$F_{n+2} = F_{n+1} + F_n \quad (7.1)$$

Выбор начальных условий для последовательности чисел Фибоначчи не важен;

существенное свойство этой последовательности определяется рекуррентным соотношением. Будем предполагать $F_0 = 0, F_1 = 1$ (иногда $F_0 = F_1 = 1$).

Рассмотрим эту задачу немного иначе.

Пара кроликов приносит раз в месяц приплод из двух крольчат (самки и самца), причем новорожденные крольчата через два месяца после рождения уже приносят приплод. Сколько кроликов появится через год, если в начале года была одна пара кроликов?

Из условия задачи следует, что через месяц будет две пары кроликов. Через два месяца приплод даст только первая пара кроликов, и получится 3 пары. А еще через месяц приплод дадут и исходная пара кроликов, и пара кроликов, появившаяся два месяца тому назад. Поэтому всего будет 5 пар кроликов. Обозначим через $F(n)$ количество пар кроликов по истечении n месяцев с начала года. Ясно, что через $n+1$ месяцев будут эти $F(n)$ пар и еще столько новорожденных пар кроликов, сколько было в конце месяца $n-1$, то есть еще $F(n-1)$ пар кроликов. Иными словами, имеет место рекуррентное соотношение

$$F(n+1) = F(n) + F(n-1) \qquad (7.2)$$

Так как, по условию, $F(0) = 1$ и $F(1) = 2$, то последовательно находим

$$F(2) = 3, F(3) = 5, F(4) = 8$$

и т.д.

В частности, $F(12) = 377$.

Числа $F(n)$ называются числами Фибоначчи. Они обладают целым рядом замечательных свойств. Теперь выведем выражение этих чисел через C_m^k. Для этого установим связь между числами Фибоначчи и следующей комбинаторной задачей.

Найти число n последовательностей,состоящих из нулей и единиц, в которых никакие две единицы не идут подряд.

Чтобы установить эту связь, возьмем любую такую последовательность и сопоставим ей пару кроликов по следующему правилу: единицам соответствуют месяцы появления на свет одной из пар "предков" данной пары (включая и исходную), а нулями - все остальные месяцы. Например, последовательность 010010100010 устанавливает такую "генеалогию": сама пара появилась в конце 11-го месяца, ее родители - в конце 7-го месяца, "дед" - в конце 5-го месяца и "прадед" - в конце второго месяца. Исходная пара кроликов тогда зашифровывается последовательностью 000000000000.

Ясно, что при этом ни в одной последовательности не могут стоять две единицы подряд - только что появившаяся пара не может, по условию, принести приплод через месяц. Кроме того, при указанном правиле различным последовательностям отвечают различные пары кроликов, и обратно, две различные пары кроликов всегда имеют разную "генеалогию", так как, по условию, крольчиха дает приплод, состоящий только из одной пары кроликов.

Установленная связь показывает, что число n -последовательностей, обладающих указанным свойством, равно $F(n)$.

Докажем теперь, что

$$F(n) = C_{n+1}^0 + C_n^1 + C_{n-1}^2 + \ldots + C_{n-p+1}^p, \qquad (7.3)$$

где $p = \frac{n+1}{2}$, если n нечетно, и $p = \frac{n}{2}$, если n четно. Иными словами, p - целая часть числа $\frac{n+1}{2}$ (в дальнейшем будем обозначать целую часть числа α через $E(\alpha)$; таким образом, $p = E(\frac{n+1}{2})$).

В самом деле, $F(n)$ - это число всех n - последовательностей из 0 и 1, в которых никакие две единицы не стоят рядом. Число же таких последовательностей, в которые входит ровно k единиц и $n-k$ нулей, равно C_{n-k+1}^k. Так как при этом должно выполняться неравенство $k \leqslant n-k+1$, то k изменяется от 0 до $E(\frac{n+1}{2})$. Применяя правило суммы, приходим к соотношению (7.3).

Равенство (7.3) можно доказать и иначе.

Положим $G(n) = C_{n+1}^0 + C_n^1 + C_{n-1}^2 + \ldots + C_{n-p+1}^p$, где $p = \frac{n+1}{2}$. Из равенства $C_n^k = C_{n-1}^k + C_{n-1}^{k-1}$ легко следует, что

$$G(n) = G(n-1) + G(n-2). \quad (7.4)$$

Кроме того, ясно, что $G(1) = 2 = F(1)$ и $G(2) = 3 = F(2)$. Так как обе последовательности $F(n)$ и $G(n)$ удовлетворяют рекуррентному соотношению $X(n) = X(n-1) + X(n-2)$, то имеем

$$G(3) = G(2) + G(1) = F(2) + F(1) = F(3),$$

и, вообще, $G(n) = F(n)$.

Другой метод доказательства

В предыдущем разделе непосредственно установлена связь между задачей Фибоначчи и комбинаторной задачей. Эту связь можно установить и иначе, непосредственно доказав, что число $T(n)$ решений комбинаторной задачи удовлетворяет тому же рекуррентному соотношению

$$T(n+1) = T(n) + T(n-1), \quad (7.5)$$

что и числа Фибоначчи. В самом деле, возьмем любую $(n+1)$ -последовательность нулей и единиц, удовлетворяющую условию, что никакие две единицы не идут подряд. Она может оканчиваться или на 0, или на 1. Если она оканчивается на 0, то, отбросив его, получим n -последовательность, удовлетворяющую нашему условию. Если взять любую n -последовательность нулей и единиц, в которой подряд не идут две единицы, и приписать к ней нуль, то получим $(n+1)$ -последовательность с тем же свойством. Таким образом доказано, что число последовательностей, оканчивающихся на нуль, равно $T(n)$.

Пусть теперь последовательность оканчивается на 1. Так как двух единиц подряд быть не может, то перед этой единицей стоит нуль. Иными словами, последовательность оканчивается на 01. Остающаяся же после отбрасывания 0 и 1 $(n-1)$ -последовательность может быть любой, лишь бы в ней не шли подряд две единицы. Поэтому число последовательностей, оканчивающихся единицей, равно $T(n-1)$. Но каждая последовательность оканчивается или на 0, или на 1. В силу правила суммы получаем, что $T(n+1) = T(n) + T(n-1)$.

Таким образом, получено то же самое рекуррентное соотношение. Отсюда еще не вытекает, что числа $T(n)$ и $F(n)$ совпадают.

Процесс последовательных разбиений

Для решения комбинаторных задач часто применяют метод, использованный в предыдущем пункте: устанавливают для данной задачи рекуррентное соотношение и показывают, что оно совпадает с рекуррентным соотношением для другой задачи, решение которой нам уже известно. Если при этом совпадают и начальные члены последовательностей в достаточном числе, то обе задачи имеют одинаковые решения.

Применим описанный прием для решения следующей задачи.

Пусть дано некоторое множество из n предметов, стоящих в определенном порядке. Разобьем это множество на две непустые части так, чтобы одна из этих частей лежала левее второй (то есть, скажем, одна часть состоит из элементов от первого до m - го, а вторая - из элементов от $(m+1)$ -го до n -го). После этого каждую из частей таким же образом разобьем на две непустые части (если одна из частей состоит уже из одного предмета, она не подвергается дальнейшим разбиениям). Этот процесс продолжается до тех пор, пока не получим части, состоящие из одного предмета каждая. Сколько существует таких процессов разбиения (два процесса считаются различными, если хотя бы на одном шагу они приводят к разным результатам)?

Обозначим число способов разбиения для множества из $n+1$ предметов через B_n. На первом шагу это множество может быть разбито n способами (первая часть может содержать один предмет, два предмета,…, n предметов). В соответствии с этим множество всех процессов

разбиений распадается на n классов - в s - класс входят процессы, при которых первая часть состоит из s предметов.

Подсчитаем число процессов в s -м классе. В первой части содержится s элементов. Поэтому ее можно разбивать далее B_{s-1} различными процессами. Вторая же часть содержит $n-s+1$ элементов, и ее можно разбивать далее B_{n-s} процессами. По правилу произведения получаем, что s - класс состоит из $B_{s-1}B_{n-s}$ различных процессов. По правилу суммы отсюда вытекает, что

$$B_n = B_0B_{n-1} + B_1B_{n-2} + \ldots + B_{n-1}B_0. \quad 7.6)$$

Таким образом получено рекуррентное соотношение для B_n. Двоичный поиск, поиск делением пополам. Поиском по числам Фибоначчи называется поиск, основанный на том, что область поиска делится в точках, являющихся числами Фибоначчи.

Задача: "Затруднение мажордома"

Бывают комбинаторные задачи, в которых приходится составлять не одно рекуррентное соотношение, а систему соотношений, связывающую несколько последовательностей. Эти соотношения выражают $(n+1)$ -у члены последовательностей через предыдущие члены не только данной, но и остальных последовательностей.

Задача: "Затруднение мажордома". Однажды мажордом короля Артура обнаружил, что к обеду за круглым столом приглашено 6 пар враждующих рыцарей. Сколькими способами можно рассадить их так, чтобы никакие два врага не сидели рядом?

Если мы найдем какой-то способ рассадки рыцарей, то, пересаживая их по кругу, получим еще 11 способов. Мы не будем сейчас считать различными способы, получающиеся друг из друга такой циклической пересадкой.

Введем следующие обозначения. Пусть число рыцарей равно $2n$. Через A_n обозначим число способов рассадки, при которых никакие два врага не сидят рядом. Через B_n обозначим число способов, при которых рядом сидит ровно одна пара врагов, и через C_n - число способов, при которых есть ровно две пары враждующих соседей.

Выведем сначала формулу, выражающую A_{n+1} через A_n, B_n, C_n. Пусть $n+1$ пар рыцарей посажены так, что никакие два врага не сидят рядом. Мы будем считать, что все враждующие пары рыцарей занумерованы. Попросим встать из-за стола пару рыцарей с номером $n+1$. Тогда возможны три случая: среди оставшихся за столом нет одной пары соседей- врагов, есть одна такая пара и есть две такие пары (ушедшие рыцари могли разделять эти пары). Мы считаем, что $n>1$. При $n=1$ последующие рассуждения теряют силу.

Выясним теперь, сколькими способами можно снова посадить ушедших рыцарей за стол так, чтобы после этого не было одной пары соседей-врагов.

Проще всего посадить их, если за столом рядом сидят две пары врагов. В этом случае один из вновь пришедших садится между рыцарями первой пары, а другой – между рыцарями второй пары. Это можно сделать двумя способами. Но так как число способов рассадки $2n$ рыцарей, при которых две пары соседей оказались врагами, равно C_n, то всего получилось $2C_n$ способов.

Пусть теперь рядом сидит только одна пара врагов. Один из вернувшихся должен сесть между ними. Тогда за столом окажутся $2n+1$ рыцарей, между которыми есть $2n+1$ мест. Из них два места - рядом с только что севшим гостем – запретны для второго рыцаря, и ему остается $2n-1$ мест. Так как первым может войти любой из двух вышедших рыцарей, то получается $2(2m-1)$ способов рассадки. Но число случаев, когда $2n$ рыцарей сели так, чтобы ровно одна пара врагов оказалась соседями, ровно B_n. Поэтому мы получаем $2(2n-1)B_n$ способов посадить гостей требуемым образом.

Наконец, пусть никакие два врага не сидели рядом. В этом случае первый рыцарь садится между любыми двумя гостями - это он может сделать $2n$ способами. После этого для его врага останется $2n-1$ мест - он может занять любое место, кроме двух мест, соседних с только что севшим рыцарем. Таким образом, если $2n$ рыцарей уже сидели нужным образом, то вернувшихся гостей можно посадить $2n(2n-1)A_n$ способами. Как уже отмечалось, разработанными случаями исчерпываются все возможности. Поэтому имеет место рекуррентное соотношение

$$A_{n+1} = 2n(2n-1)A_n + 2(2n-1)B_n + 2C. \quad 7.7)$$

Этого соотношения пока недостаточно, чтобы найти A_n для всех значений n. Надо еще узнать, как выражаются B_{n+1}, C_{n+1} через A_n, B_n, C_n.

Предположим, что среди $2n+2, n>1$ рыцарей оказалась ровно одна пара врагов-соседей. Мы знаем, что это может произойти в B_{n+1} случаях. Во избежание ссоры попросим их удалиться из-за стола. Тогда останется $2n$ рыцарей, причем возможно одно из двух: либо среди оставшихся нет врагов-соседей, либо есть ровно одна пара таких врагов - до ухода покинувших зал они сидели по обе стороны от них и теперь оказались рядом. Во втором случае ушедших можно посадить обратно только на старое место - иначе появится вторая пара враждующих соседей. Но так как $2n$ рыцарей можно посадить B_n способами так, чтобы была только одна пара враждующих соседей, то мы получаем $2B_n$ вариантов (возвратившихся рыцарей можно поменять местами). В первом же случае можно посадить ушедших между любыми двумя рыцарями, то есть $2n$ способами, а так как их еще можно поменять местами, то получится $4n$ способов. Комбинируя их со всеми способами посадки n пар рыцарей, при которых нет соседей врагов, получаем $4nA_n$ способов. Наконец, номер ушедшей и вернувшейся пары рыцарей мог быть любым от 1 до $n+1$. Отсюда вытекает, что рекуррентное соотношение для B_{n+1} имеет вид

$$B_{n+1} = 4n(n+1)A_n + 2(n+1)B_n. \quad 7.8)$$

Наконец, разберем случай, когда среди $2n+2$ рыцарей было две пары врагов-соседей. Номера этих пар можно выбрать $C^2_{n+} = \frac{n(n+1)}{2}$ способами. Заменим каждую пару одним новым рыцарем, причем будем считать новых двух рыцарей врагами. Тогда за столом будут сидеть $2n$ рыцарей, причем среди них либо не будет ни одной пары врагов-соседей (если новые рыцари не сидят рядом), либо только одна такая пара.

Первый вариант может быть в A_n случаях. Вернуться к исходной компании мы можем 4 способами благодаря возможности изменить порядок рыцарей в каждой паре. Поэтому первый вариант приводит к $4C^2_{n+1}A_n = 2n(n+1)A_n$ способами.

Второй же вариант может быть в $\frac{1}{n}B_n$ случаях. Имеется B_n случаев, когда какая-нибудь пара врагов сидит рядом. Если указать, какая именно пара должна сидеть рядом, получим в n раз меньше случаев.

Здесь тоже можно вернуться к исходной компании 4 способами, и мы получаем всего $2(m+1)B_n$ способов. Отсюда вытекает, что при $n \geqslant 1$

$$C_{n+1} = 2n(n+1)A_n + 2(n+1)B_n. \quad 7.9)$$

Мы получили систему рекуррентных соотношений

$$A_{n+1} = 2n(2n-1)A_n + 2(2n-1)B_n + 2C \quad 7.10)$$

$$B_{n+1} = 4n(n+1)A_n + 2(n+1)B_n. \quad 7.11)$$

$$C_{n+1} = 2n(n+1)A_n + 2(n+1)B_n. \quad 7.12)$$

Они справедливы при $n \geqslant 2$. Но простой подсчет показывает, что $A_2 = 2, B_2 = 0, C_2 = 4$. Поэтому из соотношений 7.10-7.12 вытекает, что $A_3 = 32, B_3 = 48, C_3 = 24$. Продолжая далее, находим, что гостей можно посадить за стол требуемым образом $A_6 = 12771840$ способами.

Задание:

1. Запрограммируйте описанные выше алгоритмы

Содержание отчета
– Название работы;
– Цель работы;
– Формулировку задания;
– Решенные задания, с пояснением решения в виде таблиц, рассуждений, рисунков.

Контрольные вопросы
1. Сформулируйте алгоритм сочетаний.
2. Сформулируйте алгоритм перемещений
3. Сформулируйте алгоритм размещения без повторений

Список литературы:

Основная литература

1. Окулов С. М. Программирование в алгоритмах. / М. Окулов. – М.: БИНОМ. Лаборатория знаний, 2002. – 341 с.
2. Иванов Б. Н. Дискретная математика. Алгоритмы и программы. – М.: Лаборатория Базовых Знаний, 2002.

Дополнительная литература

1. Семакин И. Г. Основы программирования: учебник для сред. проф. образования / И. Г. Семакин, А. П. Шестаков. – М.: Издательский центр «Академия», 2007.
2. Акимов О.Е. Дискретная математика: логика, группы, графы. – М.: Лаборатория Базовых Знаний, 2001.

Практическая работа № 14

Тема: Реализация алгоритма поиска в графе.

Цель: закрепление умений по программированию алгоритмов.

Пояснения к работе:

Нахождение кратчайшего пути на сегодняшний день является жизненно необходимой задачей и используется практически везде, начиная от нахождения оптимального маршрута между двумя объектами на местности (например, кратчайший путь от дома до университета), в системах автопилота, для нахождения оптимального маршрута при перевозках, коммутации информационного пакета в сетях и т.п.

Кратчайший путь рассматривается при помощи некоторого математического объекта, называемого графом. Поиск кратчайшего пути ведется между двумя заданными вершинами в графе. Результатом является путь, то есть последовательность вершин и ребер, инцидентных двум соседним вершинам, и его длина.

Рассмотрим три наиболее эффективных алгоритма нахождения кратчайшего пути:

- алгоритм Дейкстры;
- алгоритм Флойда;
- переборные алгоритмы.

Указанные алгоритмы легко выполняются при малом количестве вершин в графе. При увеличении их количества задача поиска кратчайшего пути усложняется.

Алгоритм Дейкстры

Данный алгоритм является алгоритмом на графах, который изобретен нидерландским ученым Э. Дейкстрой в 1959 году. Алгоритм находит кратчайшее расстояние от одной из вершин графа до всех остальных и работает только для графов без ребер отрицательного веса.

Каждой вершине приписывается вес – это вес пути от начальной вершины до данной. Также каждая вершина может быть выделена. Если вершина выделена, то путь от нее до начальной вершины кратчайший, если нет – то временный. Обходя граф, алгоритм считает для каждой вершины маршрут, и, если он оказывается кратчайшим, выделяет вершину. Весом данной вершины становится вес пути. Для всех соседей данной вершины алгоритм также рассчитывает вес, при этом ни при каких условиях не выделяя их. Алгоритм заканчивает свою работу, дойдя до конечной вершины, и весом кратчайшего пути становится вес конечной вершины.

Алгоритм Дейкстры

Шаг 1. Всем вершинам, за исключением первой, присваивается вес равный бесконечности, а первой вершине – 0.

Шаг 2. Все вершины не выделены.

Шаг 3. Первая вершина объявляется текущей.

Шаг 4. Вес всех невыделенных вершин пересчитывается по формуле: вес невыделенной вершины есть минимальное число из старого веса данной вершины, суммы веса текущей вершины и веса ребра, соединяющего текущую вершину с невыделенной.

Шаг 5. Среди невыделенных вершин ищется вершина с минимальным весом. Если таковая не найдена, то есть вес всех вершин равен бесконечности, то маршрут не существует. Следовательно, выход. Иначе, текущей становится найденная вершина. Она же выделяется.

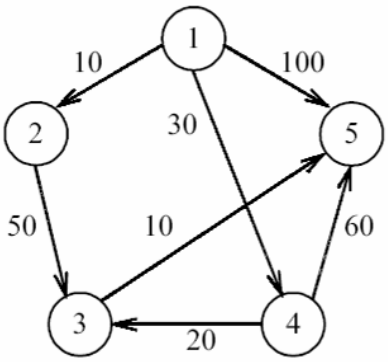
Шаг 6. Если текущей вершиной оказывается конечная, то путь найден, и его вес есть вес конечной вершины.

Шаг 7. Переход на шаг 4.

В программной реализации алгоритма Дейкстры построим множество S вершин, для которых кратчайшие пути от начальной вершины уже известны. На каждом шаге к множеству S добавляется та из оставшихся вершин, расстояние до которой от начальной вершины меньше, чем для других оставшихся вершин. При этом будем использовать массив D, в который записываются длины кратчайших путей для каждой вершины. Когда множество S будет содержать все вершины графа, тогда массив D будет содержать длины кратчайших путей от начальной вершины к каждой вершине.

Помимо указанных массивов будем использовать матрицу длин C, где элемент C[i, j] – длина ребра (i, j), если ребра нет, то ее длина полагается равной бесконечности, то есть больше любой фактической длины ребер. Фактически матрица C представляет собой матрицу смежности, в которой все нулевые элементы заменены на бесконечность.

Для определения самого кратчайшего пути введем массив P вершин, где P[v] будет содержать вершину, непосредственно предшествующую вершине v в кратчайшем пути (рис. 1).



Итерация	*S*	*w*	**D**[2]	**D**[3]	**D**[4]	**D**[5]
начало	{1}	–	10	∞	30	100
1	{1, 2}	2	10	60	30	100
2	{1, 2, 4}	4	10	50	30	90
3	{1, 2, 4, 3}	3	10	50	30	60
4	{1, 2, 4, 3,5}	5	10	50	30	60

Массив **P**:

╳	1	4	1	3

Кратчайший путь из 1 в 5: {1, 4, 3, 5}

Рис.1. Демонстрация алгоритма Дейкстры

Сложность алгоритма Дейкстры зависит от способа нахождения вершины, а также способа хранения множества непосещенных вершин и способа обновления длин.

Если для представления графа использовать матрицу смежности, то время выполнения этого алгоритма имеет порядок $O(n^2)$, где n – количество вершин графа.

Алгоритм Флойда

Рассматриваемый алгоритм иногда называют алгоритмом Флойда-Уоршелла. Алгоритм Флойда-Уоршелла является алгоритмом на графах, который разработан в 1962 году Робертом Флойдом и Стивеном Уоршеллом. Он служит для нахождения кратчайших путей между всеми парами вершин графа.

Метод Флойда непосредственно основывается на том факте, что в графе с положительными весами ребер всякий неэлементарный (содержащий более 1 ребра) кратчайший путь состоит из других кратчайших путей.

Этот алгоритм более общий по сравнению с алгоритмом Дейкстры, так как он находит кратчайшие пути между любыми двумя вершинами графа.

В алгоритме Флойда используется матрица A размером $n \times n$, в которой вычисляются длины кратчайших путей. Элемент $A[i, j]$ равен расстоянию от вершины i к вершине j, которое

имеет конечное значение, если существует ребро (i, j), и равен бесконечности в противном случае.

Алгоритм Флойда

Основная идея алгоритма. Пусть есть три вершины i, j, k и заданы расстояния между ними. Если выполняется неравенство $A[i,k]+A[k,j]<A[i,j]$, то целесообразно заменить путь $i \to j$ путем $i \to k \to j$. Такая замена выполняется систематически в процессе выполнения данного алгоритма.

Шаг 0. Определяем начальную матрицу расстояния A_0 и матрицу последовательности вершин S_0. Каждый диагональный элемент обеих матриц равен 0, таким образом, показывая, что эти элементы в вычислениях не участвуют. Полагаем k = 1.

Основной шаг k. Задаем строку k и столбец k как ведущую строку и ведущий столбец. Рассматриваем возможность применения замены описанной выше, ко всем элементам $A[i,j]$ матрицы A_{k-1}. Если выполняется неравенство $A[i,k]+A[k,j]<A[i,j]$, ($i \neq k$, $j \neq k$, $i \neq j$), тогда выполняем следующие действия:

1) создаем матрицу A_k путем замены в матрице A_{k-1} элемента $A[i,j]$ на сумму $A[i,k]+A[k,j]$;
2) создаем матрицу S_k путем замены в матрице S_{k-1} элемента $S[i,j]$ на k. Полагаем k = k + 1 и повторяем шаг k.

Таким образом, алгоритм Флойда делает n итераций, после i-й итерации матрица A будет содержать длины кратчайших путей между любыми двумя парами вершин при условии, что эти пути проходят через вершины от первой до i-й. На каждой итерации перебираются все пары вершин и путь между ними сокращается при помощи i-й вершины (рис. 2).

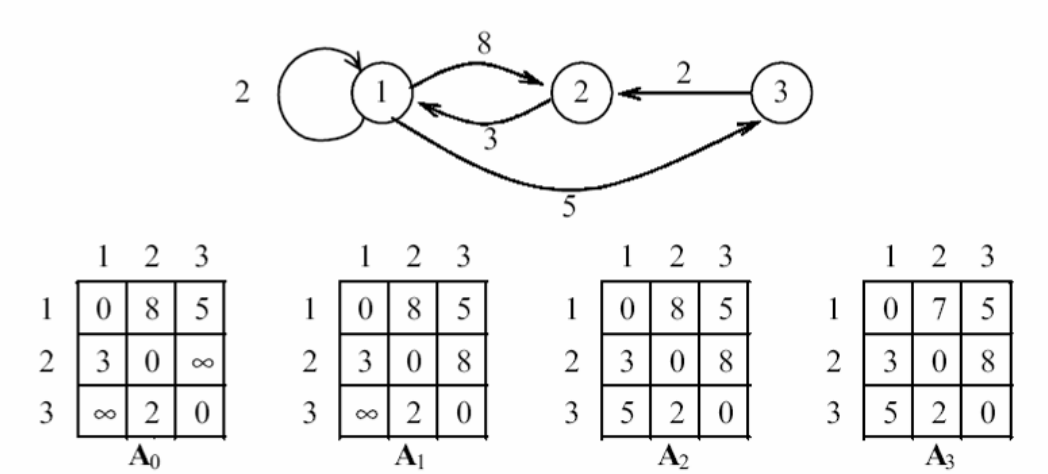
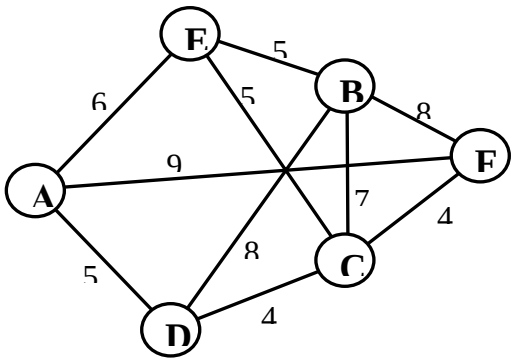


Рис.2. Демонстрация алгоритма Флойда

Заметим, что если граф неориентированный, то все матрицы, получаемые в результате преобразований симметричны и, следовательно, достаточно вычислять только элементы, расположенные выше главной диагонали.

Если граф представлен матрицей смежности, то время выполнения этого алгоритма имеет порядок $O(n^3)$, поскольку в нем присутствуют вложенные друг в друга три цикла.



Задание:

1. Реализуйте программы, в которых выполняются алгоритм Дейкстры и алгоритм Флойда.
2. Оля (A), Маша (B), Витя (C), Дима (D), Ваня (E) и Катя (F) живут в разных городах. Стоимость

билетов из разных городов известна (рис.). Добраться до городов можно разными способами. Определить наименьшую сумму, которую нужно потратить, чтобы Оля могла навестить каждого из своих друзей.

3. Квадратное озеро задается матрицей M×N и покрыто мелкими островками. В левом верхнем углу находится плот размером m×m. За один шаг плот может передвигаться на одну клетку по вертикали или горизонтали. Требуется определить кратчайший путь плота до правого нижнего угла.
4. Напишите алгоритм, находящий строку длиной 100 символов, состоящую только из букв «A», «B», «C», такую, что в ней никакие две соседние подстроки не равны друг другу. Воспользуйтесь перебором с возвратом.

Содержание отчета

– Название работы;
– Цель работы;
– Формулировку задания;
– Решенные задания, с пояснением решения в виде таблиц, рассуждений, рисунков.

Контрольные вопросы

1. С какими видами графов работают алгоритмы Дейкстры, Флойда?
2. Как от представления графа зависит эффективность алгоритма его обхода?
3. За счет чего поиск в ширину является достаточно ресурсоемким алгоритмом?
4. В чем преимущества алгоритмов обхода графа в ширину?
5. Каким образом в алгоритме перебора с возвратом при обходе графа обрабатывается посещение тупиковых вершин?

Список литературы:

Основная литература

1. Окулов С. М. Программирование в алгоритмах. / М. Окулов. – М.: БИНОМ. Лаборатория знаний, 2002. – 341 с.
2. Иванов Б. Н. Дискретная математика. Алгоритмы и программы. – М.: Лаборатория Базовых Знаний, 2002.

Дополнительная литература

1. Семакин И. Г. Основы программирования: учебник для сред. проф. образования / И. Г. Семакин, А. П. Шестаков. – М.: Издательский центр «Академия», 2007.
2. Акимов О.Е. Дискретная математика: логика, группы, графы. – М.: Лаборатория Базовых Знаний, 2001.