Учреждение образования

«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ

ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Кафедра информационных систем и технологий

**Математическое программирование**

**Курс лекций**

**для студентов специальности:**

1-40 01 02-03 – «Информационные системы и технологии (издательско-полиграфический комплекс)»

Составитель:

Доцент каф. ИСиТ,

кандидат технических наук

А. И. Бракович

Минск 2017

**Лекция 1**

4:

**ВВЕДЕНИЕ В МАТЕМАТИЧЕСКОЕ ПРОГРАММИРОВАНИЕ**

|  |  |
| --- | --- |
| Лекции | **34 часа (**17 лекций**)** |
| Лабораторные работы | **18 часов (**9 лабораторных работ**)** |
| Самостоятельная работа | **78 часов (**выполнение лабораторных**)** |
| **Всего** | **130 часов** |
| Экзамен | **2 семестр** |

**5: Математическое программирование** –область математики, разрабатывающая теорию и численные методы решения многомерных экстремальных задач, т.е. задач на экстремум функции многих переменных с ограничением на область определения.

Решение задачи математического программирования осуществляется в 4 этапа.

1. Построение математической модели.
2. Классификация задачи.
3. Выбор метода решения.
4. Вычисление.

6: В общем виде **модель задачи математического программирования** выглядит следующим образом:

|  |  |
| --- | --- |
| , , где | |
|  | – искомая, в общем случае векторная, величина; |
|  | – область определения искомой величины; |
|  | – функция цели (функция определяющая значение критерия оптимальности); |

В зависимости от природы множества  и вида функции  задачи математического программирования **классифицируются** как задачи

* дискретного программирования (комбинаторная оптимизация) ( конечно или счетно);
* целочисленного программирования ( подмножество множества целых чисел);
* линейного программирования (– линейная функция,  – может быть определено с помощью линейных неравенств);
* нелинейного программирования (– нелинейная функция и/или в описании  присутствует хотя бы одна нелинейная функция);
* векторная оптимизация (– векторная функция).

Кроме того, разделами математического программирования являются динамическое, стохастическое и параметрическое программирование, сетевое планирование, потоки в сетях и т.д.

**Метод решения** задачи математического программирования определятся в зависимости от исходных данных.

**Вычисление** решения задачи математического программирования осуществляется, как правило, с помощью компьютерной техники.

7: **СМЕЖНЫЕ ДИСЦИПЛИНЫ**

8: **РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА**

|  |  |
| --- | --- |
| 1. | Смелов, В. В. Комбинаторные алгоритмы оптимизации : учеб.-метод. пособие для студентов специальности «Информационные системы и технологии» (издательско-полиграфический комплекс) / В. В. Смелов, А. И. Бракович – Минск : БГТУ, 2010. |
| 2. | Смелов, В. В. Основы сетевого планирования: учеб.-метод. пособие для студентов специальности «Информационные системы и технологии (издательско-полиграфический комплекс»/ В. В. Смелов, Т. П. Брусенцова. – Минск: БГТУ, 2010. – 231 с. |
| 3. | Смелов, В. В. Алгоритмы на графах и их реализации на С++ / В. В. Смелов, Л. С. Мороз. – Минск: БГТУ, 2011. – 144 с. |
| 4. | Костевич Л.С. Математическое программирование. – Мн.: Новое знание, 2003, – 424 с. |
| 5. | Таха Х.А. Введение в исследование операций. – М.: Вильямс, 2001. – 912 с. |
| 6. | Кузнецов А.В., Сакович В.А., Холод Н.И. Высшая математика. Математическое программирование. –Мн.: Высш.шк., 1994. – 288 с. |

**9: ПРИМЕРЫ ЗАДАЧ МАТЕМАТИЧЕСКОГО ПРОГРАММИРОВАНИЯ**

* Задача о кратчайшем расстоянии между вершинами графа
* Задача о рюкзаке
* Задача о коммивояжере (о бродячем торговце)
* Задача о нахождении максимального потока в сети
* Задача линейного программирования
* Транспортная задача
* Задача нелинейного программирования
* Векторная оптимизация
* Сетевое планирование

10: **Вспомогательные функции**

Алгоритмы, в основном мы будем оформлять в виде функции. Целью первой лабораторной работы является приобретение навыков составления и отладки программ с использованием пользовательских функций для замера продолжительности процесса вычисления.

**Генерация случайных чисел**

Директивы препроцессора представляют собой инструкции, записанные в исходном тексте программы и предназначенные для выполнения препроцессором языка. Фактически это часть компилятора, которая умеет исполнять директивы. #include вставляет текст файла, указанного далее.

Назначение функции srand – установка начального значения псевдослучайного числа.

Функция rand возвращает псевдослучайное целоче число от 0 до RAND\_MAX. RAND\_MAX это положительная константа, определенная с помощью директивы # include <cstdlib>.

11: **Функции времени**

Функция time\_t возвращает количество секунд прошедшие с 00:00:00 01.01.1970 к точке вызова или/и в буфер, если параметр t не NULL.

Функция clock\_t возвращает количество единиц процессорного времени или тактовую частоту (сек = CLOCKS\_PER\_SEC единиц) прошедших с момента старта приложения. Константа CLOCKS\_PER\_SEC показывает сколько единиц процессорного времени находится в одной секунде.

**Лекция 2**

**КОМБИНАТОРНЫЕ МЕТОДЫ РЕШЕНИЯ ОПТИМИЗАЦИОННЫХ ЗАДАЧ**

4:

1. **Генерация подмножеств заданного множества**:

- разработка генератора подмножеств на С++;

- решение задачи о рюкзаке.

1. **Генерация сочетаний:**

- разработка генератора сочетаний на С++;

- решение задачи об оптимальной загрузке.

1. **Генерация перестановок:**

- разработка генератора перестановок на С++;

- решение задачи о коммивояжере.

1. **Генерация размещений:**

- разработка генератора сочетаний на С++;

- решение задачи об оптимальной загрузке (с центровкой)

**Особенность алгоритмов:**

1. Невозможно использовать для задач большой размерности.
2. Применяются тогда, когда требуется:

- точное решение или достаточное решение;

- размерность задачи небольшая.

1. Сложность этих алгоритмов:

- является верхней оценкой сложности решения задач;

- не зависит от данных;

- улучшение возможно только в статистическом смысле.

5:

***Комбинаторный анализ (комбинаторика, комбинаторная математика)*** – раздел математики, посвященный решению задач выбора и расположения элементов некоторого, обычно конечного, множества в соответствии с заданными правилами.

1. **Генерация подмножеств заданного множества**

6: Пусть  – конечное множество мощности  **Множество всех подмножеств** множества  называется ***булеаном***  и обозначается  Количество элементов булеана множества  вычисляется по формуле

 (1)

Алгоритм генерации множества всех подмножеств основывается на взаимно однозначном соответствии между элементами булеана  множества  и всеми целыми числами множества , записанными в двоичном виде. Следует сказать, что алгоритм имеет сложность  и поэтому реально может применяться для множеств с небольшой мощностью. Например, генерация булеана множества мощностью  на компьютере с тактовой частотой процессора 3 ГГц займет более 100 лет.

7: Построение элементов булеана множества  сводится к следующему алгоритму:

1. Пронумеровать элементы заданного множества  начиная с нуля.
2. Сформировать битовую последовательность  состоящую из  двоичных нулей. Пронумеровать элементы этой последовательности справа налево, начиная с нуля.
3. Последовательно выполнить шаги 4 и 5 алгоритма  раз.
4. Выбрать из множества  элементы с номерами  для которых  Полученное подмножество будет являться элементом булеана  В первом случае не будет выбран ни один элемент (пустое подмножество) множества  так как исходная последовательность  состоит только из нулей.
5. Интерпретируя битовую последовательность как целое положительное число, увеличить это число на единицу.

8: На рис. представлен пример построения булеана для множества  Во втором сверху прямоугольнике изображены элементы     множества  Над ними указаны их номера:  для   для   для  и  для 

Под элементами множества  изображен столбец битовых последовательностей. Каждая строка столбца соответствует состоянию последовательности  при очередном прохождении шага 4 алгоритма. Слева от столбца указаны целые числа, которые являются интерпретацией битовых последовательностей как целых чисел в десятичной системе счисления.

Стрелки на рис. указывают на строки другого столбца, который содержит элементы множества  соответствующие битовым последовательностям. Каждая строка этого столбца содержит элементы одного из подмножеств множества 

9: Разберем еще один пример: Требуется получить множество всех подмножеств: . По формуле 1 узнаем количество всех подмножеств – 24 = 16.

10-12: **Принципы реализации генератора на С++**

1. Все генераторы должны иметь одинаковый интерфейс
2. Должна быть возможность применения нескольких генераторов одновременно.
3. Функции должны легко встраиваться в другие.

На рис. 2 и 3 представлена реализация генератора множества всех подмножеств на языке C++. Генератор реализован в виде структуры **subset**. Применение структуры позволяет создавать несколько экземпляров одного генератора и упростить его вызов в других программах.

Структура **subset**  имеет единственный конструктор, параметром которого является мощность исходного множества.

Для хранения текущего состояния генератора используются три переменные: **n** (мощность исходного множества, которая не должна превышать 63), **sn** (текущееколичество элементов массива индексов), **sset** (адрес нулевого элемента массива индексов) и **mask** (битовая последовательность). Первоначальное значение всех переменных инициализируются конструктором. Причем значения **n** и **sset** послеинициализацииостаются неизменными, а значения остальных переменных меняются в процессе работы генератора.

Помимо конструктора, структура содержит еще пять функций-членов.

Функция **getfirst** позволяет заполнить массив индексов в соответствии с текущим значением битовой последовательности **mask**. Функция устанавливает значение **sn**, равным количеству двоичных единиц в последовательности **mask**, а в массив **sset**,начиная с нулевого элемента, записывает номера единичных позиций (позиции нумеруются справа налево) в последовательности **mask**. Вызов функции сразу после создания структуры приводит к установке значения **sn** в нуль, что соответствует битовой последовательности **mask**, состоящей из одних нулей (конструктор инициализирует **mask** нулевым значением). Функция **getfirst** всегда возвращает значение **sn**. Обычно пользователь эту функцию вызывает один раз – для формирования первого (пустого) подмножества. Для получения остальных подмножеств применяется функция **getnext**.

Функция **getnext** увеличивает значение **mask** на единицу и вызывает функцию **getfirst**, которая формирует новый массив индексов и **sn** в соответствии с новым значением **mask**. Функция **getnext** возвращает значение **sn**, если массив индексов сформирован. Последний вызов функции возвращает значение – 1.

Функция **ntx** возвращает значение элемента массива индексов по индексу этого элемента и служит для сокращения записи при переборе элементов массива.

Функция **count** вычисляет и возвращает количество элементов булеана заданного множества.

Для перевода генератора в начальное состояние служит функция **reset**. После вызова **reset** снова могут вызываться **getfirst** и **getnext**.

13-15: На рис. приведен пример применения генератора множества всех подмножеств и результат выполнения программы.

В качестве исходного множества в примере используется строковый массив, состоящий из четырех строк. Вначале этот массив распечатывается. Затем объявляется структура **subset** и ей в качестве параметра передается количество элементов исходного множества (значение ).

Цикл формирования подмножеств начинается функцией **getfirst**,которая формирует первый (пустой) массив **sset**, соответствующий пустому подмножеству. Все остальные массивы индексов (для непустых подмножества) формируются функцией **getnext**. Выход из цикла происходит после того, как **getnext** возвращает отрицательное значение (получены все подмножества). Выбор элементов исходного массива осуществляется с помощью функции **ntx**.

**Сложность: **

16: **Решение упрощенной задачи о рюкзаке с помощью генератора множества всех подмножеств**

Задача о рюкзаке является одной из известных задач, решаемых методом перебора. Сформулируем упрощенную задачу о рюкзаке.

Существует  различных предметов, характеризующихся объемом  и стоимостью , . Необходимо выбрать несколько разных предметов таким способом, чтобы они поместились в рюкзаке объемом  и при этом их суммарная стоимость была максимальной.

Математическая модель задачи может быть записана следующим образом:

   

где  – неизвестные, которые требуется найти.

Решением задачи при такой постановке будет вектор . Каждый элемент  вектора может принимать значение  или 1. При этом если  то -ый предмет не выбран, и если  то -й предмет выбран для размещения в рюкзаке.

17: Задача имеет следующие исходные данные:

 – вместимость (объем) рюкзака;

 – количество предметов;

 – вектор объемов предметов;

 – вектор стоимостей предметов.

С помощью генератора всех подмножеств последовательно генерируются все массивы индексов (на рис. 6 второй слева столбец), соответствующие битовой последовательности (первый столбец слева). На основе массивов индексов формируются все возможные подмножества предметов. Для каждого подмножества вычисляется суммарный объем (столбец с заголовком) и сверяется с вместимостью рюкзака. Если выбранное подмножество предметов помещается в рюкзак (суммарный объем предметов не превышает вместимости рюкзака ), то для выбранных предметов рассчитывается суммарная стоимость (столбец ). Окончательным решением задачи будет подмножество предметов, имеющее максимальную суммарную стоимость при допустимом суммарном объеме. На рис. 6 строка, соответствующая решению, отмечена рамкой.

18-23: Пример реализации функции **knapsack\_s** на языке C++, которая решает задачу о рюкзаке.

Функция **knapsack\_s** имеет четыре входных параметра, задающих условие задачи: **V** (объем рюкзака), **n** (количество предметов), **v** (массив размерностью **n**, содержащий объемы всех предметов), **c** (массив размерностью **n**, содержащий стоимости всех предметов), а также один выходной параметр **m** (массив размерностью **n**). Каждый элемент массива **m** может быть только единицей или нулем. Единица указывает, что соответствующий предмет включен, а ноль – не включен в оптимальный перечень предметов. В результате выполнения функция возвращает оптимальную стоимость рюкзака, т. е. максимальную суммарную стоимость предметов, которые можно одновременно поместить в рюкзак заданной вместимости.

В процессе своей работы функция **knapsack\_s** использует генератор множества всех подмножеств (**combi::subset**) и вызывает три вспомогательные функции: **calcv**,  **calcc** и **setm**.

Для подключения генератора в заголовочный файл функции **knapsack\_s** добавлена директива **include**, которая включает файл **Combi.h**, содержащий шаблон структуры **combi::subset**.

Функция **calcv** предназначена для вычисления суммарного объема текущего подмножества предметов. Она принимает два входных параметра: **s** (структуру **combi::subset**) и **v** (массив размерностью **n**, содержащий объемы всех предметов), а также возвращает суммарный объем текущего подмножества предметов.

Функция **calcс** позволяетвычислить суммарную стоимость текущего подмножества предметов. Онапринимает два входных параметра: **s** (структуру **combi::subset**) и **c** (массив размерностью **n**, содержащий стоимости всех предметов), а возвращает суммарную стоимость текущего подмножества предметов.

Функция **setm** принимает два параметра: входной **s** (структуру **combi::subset**) и возвращаемый **m (**массив размерностью **n**, содержащий нули и единицы).

Функция **knapsack\_s** перебирает все подмножества множества предметов (функции **getfirst** и **getnext** генератора), вычисляет суммарный объем (функция **calcv**) каждого подмножества, для подмножества с суммарным объемом, меньшим **V**, рассчитывает суммарную стоимость (функция **calcc**) и запоминает (и возвращает) оптимальный вариант (формирует выходной массив **m** спомощьюфункции **setm**).

24-26: Оценить зависимость продолжительности вычисления оптимальной комбинации предметов от их общего количества можно с помощью следующей программы.

Для вычисления продолжительности выполнения функции **knapsack\_s** в программе используется стандартная функция **clock**, возвращающая количество условных единиц времени, прошедших с момента запуска программы. Разница между возвращаемыми значениями функции **clock**,полученными до вызова **knapsack\_s** и после, позволяет оценить продолжительность выполнения этой функции.

27: График, построенный по результатам выполнения программы. Несложно заметить, что с увеличением количества предметов (параметр **n** функции **knapsack\_s**) наединицу продолжительность выполнения функции **knapsack\_s** удваивается. Такой результат согласуется с оценкой  сложности алгоритма генерации булеана множества мощности **n**.

Эксперимент проводился на компьютере с процессором Pentium V с тактовой частотой 3,2 ГГц. Простая экстраполяция результатов этого эксперимента дает следующие результаты: для 30 предметов продолжительность вычисления будет составлять более 24 мин., для 40 – более 16 сут., а для 45 – почти два года.

28: 2. **Генерация сочетаний**

Булеан  можно рассматривать как объединение всевозможных сочетаний, построенных из элементов множества   Поэтому генерация множества  может быть сведена к генерации булеана и выбору из него всех подмножеств с мощностью 

**** (2)

** **

29: Схема построения множества сочетаний  из элементов множества  Закрашенным прямоугольником на рисунке обозначены номера (индексы) элементов битовых последовательностей   и элементов множества  Стрелки связывают битовые последовательности, содержащие три двоичные единицы и сгенерированные сочетания множества  Для каждой стрелки указаны индексы единичных позиций соответствующих битовых последовательностей. Эти индексы используются для выбора элементов из множества для включения в соответствующее сочетание. Очевидно, что такой алгоритм генерации сочетаний имеет сложность  как и алгоритм генерации множества всех подмножеств.

30: Рассмотрим еще один пример. Имеется множество всех подмножеств: . Всего подмножеств 16. Требуется найти все сочетания по три.

31-36: Представлена реализация генератора сочетаний на языке С++. Генератор реализован в виде структуры **xcombination**.

Структура **xcombination** имеет один конструктор с двумя параметрами. Первый параметр определяет количество элементов в исходном множестве, второй – количество элементов в генерируемых сочетаниях.

Для хранения текущего состояния генератора используются три переменные: **n** (мощность исходного множества), **m** (количество элементов в генерируемых сочетаниях), **sset** (адрес нулевого элемента массива индексов) и **nc** (номер текущего сочетания). Все переменныеинициализируются в конструкторе. Значение **nc** увеличивается на единицу после формирования очередного сочетания, а значение остальных переменных остается постоянным.

Кроме конструктора, структура **xcombination** содержит еще пять функций.

Функция **getfirst** не имеет параметров и предназначена для проверки корректности параметров, заданных в конструкторе. Эта функция не формирует массива индексов, как это происходило в структуре **subset** (генератор множества всех подмножеств). В основном она существует для унификации интерфейсов всех генераторов. Функция возвращает отрицательное значение, если параметры генератора заданы неверно.

Функция **getnext** формирует массив индексов следующего сочетания и увеличивает значение переменной **nc** на единицу. При каждом вызове функции для текущего массива индексов вычисляется новое значение *j*-индекса и, если оно не превышает **m**, строится новый массив индексов. При достижении *j*-индексом значения, равного или превышающего **m**, функция возвращает отрицательное значение, в других случаях возвращается положительное значение.

Функция **ntx** возвращает значение элемента массива индексов по индексу этого элемента и служит для сокращения записи при переборе элементов массива.

Функция **count** вычисляет и возвращает общее количество сочетаний из **n** по **m**.

Как и в генераторе множества всех подмножеств, для сброса генератора сочетаний в начальное состояние служит функция **reset**. После вызова **reset** снова могут вызываться **getfirst** и **getnext**.

В качестве исходного множества в примере используется строковый массив, состоящий из пяти элементов. Вначале программы этот массив распечатывается. Далее объявляется структура **xcombination** и ее конструктору в качестве параметров передаются количество элементов исходного множества и размерность сочетаний.

Генерация сочетаний начинается функцией **getfirst**.Если функция возвращает положительное значение, то сформирован индекс массивов первого сочетания. Массив индексов каждого следующего сочетания формируется функцией **getnext**. Выбор элементов исходного массива осуществляется с помощью функции **ntx**. Признаком завершения цикла генерации является отрицательное значение функции **getnext**.

37: **Решение задачи об оптимальной загрузке судна на основе генератора сочетаний**

Применение генератора сочетаний продемонстрируем на решении задачи об оптимальной загрузке судна. Сформулируем условие задачи.

На палубе судна имеется  мест для размещения стандартных контейнеров. Выбрать *n* контейнеров для погрузки на судно можно из  имеющихся в наличии. Каждый контейнер  характеризуется весом  и доходом   от его перевозки. Необходимо выбрать  контейнеров таким образом, чтобы их общий вес не превышал  но при этом доход от перевозки был максимально возможным.

Математическая модель задачи может быть записана следующим образом:

    ,

где  – неизвестные (номера выбранных контейнеров), которые требуется найти.

Решением задачи будет вектор  Каждый элемент этого вектора может принимать целое значение из отрезка  и при этом все значения  должны быть разными.

38: Схема решения задачи с применением генератора подмножеств. Задача имеет следующие исходные данные:

 – ограничение по общему весу контейнеров;

 – количество контейнеров;

 – количество свободных мест на палубе;

 – вес контейнеров;

 – доход от перевозки контейнеров.

Строки таблицы, озаглавленной символом  представляют собой все сочетания по три из множества . Эти сочетания могут быть получены с помощью соответствующего генератора. Несложно убедиться, что количество строк составляет  а порядок их перечисления соответствует порядку генерации сочетаний, рассмотренным выше алгоритмом.

Используя элементы сгенерированных сочетаний в качестве индексов для массивов  (вес каждого контейнера) и  (доход от перевозки), осуществляется выбор соответствующих значений (вторая таблица слева) , что позволяет рассчитать вес (столбец ) и доход от перевозки (столбец ) комбинации контейнеров. Решением задачи будет сочетание контейнеров, имеющее максимальный суммарный доход при допустимом суммарном весе. Строка, соответствующая решению, отмечена рамкой.

39-44: Пример реализации на языке С++ функции **boat**,решающей задачу об оптимальной загрузке судна.

Функция **boat** имеет пять входных параметров, определяющих условие задачи: **V** (максимальный допустимый суммарный вес контейнеров), **m** (количество мест на палубе для установки контейнеров), **n** (общее количество контейнеров), **v** (массив размерностью **n**, содержащий вес каждого контейнера), **c** (массив размерностью **n**, содержащий доход от перевозки каждого контейнера), а также один возвращаемый параметр **r** (массив размерностью **m**, содержащий номера выбранных контейнеров). В том случае, если решение существует, функция **boat** возвращает положительное значение, иначе – нуль.

В процессе своей работы функция **boat** использует генератор сочетаний (**combi::xcombination**) и вызывает три вспомогательные функции: **boatfnc::calcv** (расчет веса текущего сочетания контейнеров), **boatfnc::calcс** (расчет дохода от транспортировки текущего сочетания контейнеров) и **boatfnc::copycomb** (копирование текущей компинации).

Функция **boat** последовательно генерирует все возможные сочетания по **m** контейнеров, вычисляет для каждого сочетания суммарный вес(функция **boatfnc::calcv**), для сочетаний с весом, не превышающим допустимое значение **V**,вычисляетдоход от перевозки этих контейнеров (**boatfnc::calcс**), фиксирует оптимальную комбинацию контейнеров (**boatfnc::copycomb**)и возвращает оптимальную доходность или нуль, если решения нет.

45-47: Представлена программа, с помощью которой можно оценить продолжительность решения задачи о загрузке судна при разном количестве контейнеров. В программе фиксируется значение параметра **m** (количество мест для контейнеров) и вычисляется продолжительность работы функции boat в зависимости от параметра n (общее количество контейнеров).

Как и в задаче о рюкзаке, для вычисления продолжительности выполнения функции **boat** в программе, применяется стандартная функция **clock**.

48: Вид графика согласуется с оценкой  сложности алгоритма генерации сочетаний по 6 элементов из множества мощности 

Сложность: в общем случае **.**

**Лекция 3**

**КОМБИНАТОРНЫЕ МЕТОДЫ РЕШЕНИЯ ОПТИМИЗАЦИОННЫХ ЗАДАЧ**

**3. Генерация перестановок**

5: Наиболее известным методом построения множества  всех перестановок конечного множества  является ***алгоритм Джонсона – Троттера***. Алгоритм подразумевает, что все элементы множества  можно единственным способом перечислить в порядке возрастания. Отметим, что для конечного множества такой порядок всегда можно установить. Каждый элемент исходного множества  помечается специальным символом – стрелкой, которая может быть направлена влево или вправо. Первая перестановка в алгоритме Джонсона – Троттера выглядит следующим образом:



В алгоритме используется понятие ***мобильного элемента***. Элемент  последовательности элементов множества  называется мобильным, если соответствующая ему стрелка указывает на меньший соседний элемент. В первой перестановке все элементы кроме самого левого являются мобильными.

Построение множества всех перестановок с помощью алгоритма Джонсона – Троттера сводится к следующей процедуре:

1. Построить первую перестановку. Первая перестановка – это последовательность всех элементов множества  перечисленных в порядке возрастания. Стрелки всех элементов последовательности направлены влево.
2. Найти наибольший мобильный элемент в текущей перестановке. Если в последовательности нет мобильного элемента, то построены все перестановки элементов множества  – алгоритм закончил свою работу.

3. Поменять местами наибольший мобильный элемент и элемент, на который указывает стрелка наибольшего мобильного элемента.

4. Найти все элементы, большие, чем наибольший мобильный элемент (если они есть) и изменить их стрелки на противоположное направление.

5. Перейти к пункту 2.

6: Схема алгоритма генерации множества всех перестановок множества  приведена на рис.

Второй слева столбец на рис. – массив индексов, генерируемый с помощью алгоритма Джонсона – Троттера. Каждый массив индексов является одной из перестановок элементов множества . Количество массивов составляет:



Самый правый столбец (обозначен ) – множество всех перестановок элементов  Перестановки формируются как результат индексации массива 

7-13: Реализация генератора перестановок на языке C++

Генератор реализован в виде структуры **permutation**. Структураимеет один конструктор. С помощью параметра конструктору передается размерность исходного множества.

Состояние генератора определяется значениями четырех переменных: **n** (количество элементов в исходном множестве), **sset** (указатель на массив индексов), **dart** (указатель на массив стрелок), **np** (номер текущей перестановки).Всепеременные инициализируются в конструкторе. Значение **np** увеличивается на единицу после генерации очередной перестановки, значение остальных переменных остается неизменным. Элементы массивов **sset** и **dart** меняются при каждом цикле работы генератора в соответствии с алгоритмом Джонсона –Троттера.

Кроме конструктора структура **permutation** содержит еще пять функций.

Функция **getfirst** не имеет параметров и предназначена для формирования первой перестановки, которая представляет собой упорядоченную последовательность **n** неотрицательных целых чисел.

Функция **reset** позволяет сбросить текущее состояние генератора для того, чтобы начать его работу сначала. Работа функции сводится к вызову функции **getfirst**. Функция **reset** реализована главным образом для унификации интерфейсов всех генераторов.

Функция **getnext** формирует массив индексов следующей перестановки и увеличивает значение переменной **np** на единицу. При каждом вызове функции в массиве **sset** отыскивается максимальный мобильный элемент и элемент, на который указывает его стрелка, эти элементы меняются местами. Если существуют элементы **sset**,большие, чем найденный мобильный, то соответствующие им стрелки инвертируются.

Функция **ntx** возвращает значение элемента массива индексов по индексу этого элемента и служит для сокращения записи при переборе элементов массива.

Функция **count** вычисляет и возвращает общее количество перестановок **n** элементов множества.

14: **Решение задачи коммивояжера c использованием генератора перестановок**

Задача коммивояжера как и задача о рюкзаке, является классической задачей, решаемой с помощью перебора. Сформулируем условие задачи.

Коммивояжер (бродячий торговец) должен найти минимальный кольцевой маршрут обхода  городов. Расстояние  между каждой парой   городов считается известным.

Математическая модель задачи может быть записана следующим образом:

 

 

где  – неизвестные (номера выбранных городов), которые требуется найти.

15: Решением задачи будет вектор . Каждый элемент этого вектора может принимать целое значение из отрезка . При этом все значения  должны быть разными.

Решение задачи сводится к генерации всех допустимых векторов , вычислению функции  и выбору вектора, соответствующего минимальному значению функции.

16: На рис. изображена схема решения задачи коммивояжера с применением генератора перестановок. Задача решается для пяти городов.

Расстояние между городами задается матрицей *А*:

Элемент  матрицы  определяет расстояние между городами  и  где  Факт отсутствия пути из города  в город  обозначается значением  (бесконечность) соответствующего элемента.

Замечания:

1. При построении оптимального маршрута коммивояжера выбор стартового (он же является и конечным, так как маршрут кольцевой) города никак не влияет на конечный результат.
2. Если задано  городов, то перебор следует осуществлять только для  городов, поскольку стартовый город можно зафиксировать.

На рис. в качестве стартового города выбран город с номером . Поэтому перебор маршрутов осуществляется для городов с номерами 1, 2, 3, 4.

Левая часть схемы на рис. практически идентична левой части схемы на генератора перестановок. Главное отличие заключается в том, что в качестве исходного массива выбран  Причем перестановкам подлежит только внутренняя, заключенная между нулями, часть исходного массива.

В правой части схемы изображены все возможные кольцевые маршруты, которые образованы из исходного массива путем перестановок всех, кроме обрамляющих нулей, элементов. Количество маршрутов  равно количеству перестановок из четырех городов. Для каждого маршрута в округлой рамке указана длина. Длина кольцевых маршрутов, для которых не могут быть построены в силу отсутствия пути хотя бы между одной парой городов, на схеме обозначена символом  Оптимальный маршрут длиной  единиц выделен затемненной рамкой.

17-22: Пример реализации на C++ функции **salesman**, вычисляющей оптимальный кольцевой маршрут коммивояжера. Функция имеет два входных параметра: **n** (количество городов) и **d** (двумерный массив размерностью **n** на **n**, содержащий элементы матрицы расстояний), а также один возвращаемый параметр **r** (массив размерностью **n**, содержащий оптимальный маршрут).

В функции **salesman** применяется генератор перестановок (**combi::permutation**). Кроме того она вызывает шесть вспомогательных функций: **indx** (формирование перестановки городов на основе массива идексов), **distance** (вычисление длины кольцевого маршрута), **copypath** (копирование маршрута), **source** (формирование исходного массива), **firstpath** (формирование первого маршрута) и **sum** (суммирование двух чисел с учетом того, что одно из них может быть равно бесконечности).

Функция **salesman** в цикле генерирует все возможные кольцевые маршруты, вычисляет для каждого маршрута длину (функция **distance**), фиксирует оптимальный маршрут (функция **copypath**) и возвращает длину оптимального пути или значение **INF**, что обозначает отсутствие кольцевых маршрутов.

23-25: Представлена программа, позволяющая оценить продолжительность решения задачи коммивояжера в зависимости от количества городов.

В программе применяются функции **auxil::start** и **auxil::iget**, позволяющие сгенерировать расстояния между городами случайным образом, которые берутся из первой лабораторной работы.

26: Вид графика вполне согласуется с оценкой  сложности алгоритма генерации перестановок  элементов.

Сложность .

27: **4. Генерация размещений**

Множество размещений  из  по  можно получить, перестановкой элементов всех сочетаний множества . Другими словами, для получения множества размещений  требуется сначала сгенерировать все сочетания по  элементов из множества  а затем все перестановки элементов для каждого сочетания.

Генерация размещений осуществляется в три этапа:

1. Для множества  формируется множество всех сочетаний по три элемента. Таких сочетаний будет 

2. Для элементов каждого сочетания ,  генерируется все перестановки  Каждому сочетанию будет соответствовать  перестановок. Таким образом, всего будет построено  перестановок, которые в совокупности представляют собой все размещения множества 

3. Применяя полученные на предыдущем шаге размещения в качестве индексов для элементов множества  формируется множество  всех размещений по 3.

28: На рис. представлена схема построения множества размещений  из элементов множества 

Для того чтобы проследить этапы генерации множества  на схеме, ее следует рассматривать слева направо. В крайней левой позиции изображено множество , на основе которого формируются четыре сочетания.

Затем каждое сочетание рассматривается как отдельное множество, состоящее из трех элементов. Для каждого множества формируется по  перестановок. В итоге в третьем слева столбце на схеме отображены  перестановки множества 

На последнем этапе формируется множество  всех перестановок элементов множества  Элементы множества  на рис. 1 отображены в крайнем справа столбце.

29-30: Пример. Имеется множество из 4-х элементов. Необходимо получить все возможные размещения по 2 элемента. Сначала получаем множество всех подмножеств – 16. Затем берем все сочетания по 2 – их будет 6, а потом получившиеся сочетания переставляем – их получается 12.

31-37: Реализация генератора размещений на языке С++. Представлена реализация программы генератора перестановок, а на рис. 4 и 5 приведен пример построения множества размещений с помощью этой программы и результат выполнения программы соответственно.

Как и в предыдущих случаях, генератор размещений тоже реализован в виде структуры. Структура **accomodation** имеет один конструктор. С помощью двух параметров конструктору передается размерность исходного множества (параметр **n**) и размерность генерируемых размещений (параметр **m**).

В своей работе генератор размещений использует два встроенных генератора: сочетаний (указатель **cgen**) и перестановок (указатель **pgen**).

Текущее состояние генератора определяется состоянием используемых генераторов, а также значением четырех переменных: **n** (количество элементов исходного множества), **m** (размерность генерируемых размещений), **sset** (указатель на массив индексов) и **na** (номер текущего размещения). Все переменные, включая указатели генераторов, инициализируются в конструкторе. Значение **na** увеличивается на единицу после генерации очередного размещения. Элементы масcива **sset** меняются при каждом цикле работы генератора.

Кроме конструктора, структура **accomodation** содержит еще пять функций.

Функция **getfirst** не имеет параметров и предназначена для формирования первого размещения. Первое размещение совпадает с первым сочетанием, сформированным функцией **getfirst** встроенного генератора сочетаний.

Функция **reset** позволяет сбросить текущее состояние генератора для того, чтобы начать его работу сначала. Функция выполняет сброс встроенных генераторов (функция **reset**), устанавливает значение нумератора размещений (переменная **na**) в нуль и выполняет функцию **getfirst** встроенногогенератора сочетаний.

Функция **getnext** формирует массив индексов следующего размещения и увеличивает значение переменной **na** на единицу. В своей работе функция использует функции **getnext** или **getfirst** встроенных генераторов, а также функцию **getfirst** генератора размещений.

Функция **ntx** возвращает значение элемента массива индексов по индексу этого элемента и служит для сокращения записи при переборе элементов массива.

Функция **count** вычисляет и возвращает общее количество размещений из **n** по **m** элементов.

38: Решение задачи об оптимальном размещении контейнеров на судне с помощью генератора размещений. Изменим формулировку задачи об оптимальной загрузке судна и продемонстрируем способ ее решения с помощью генератора размещений. Новая формулировка следующая.

На палубе судна имеется  мест для размещения грузовых контейнеров. Выбрать  контейнеров для погрузки на судно можно из  имеющихся в наличии. Каждый контейнер  характеризуется весом  и доходом   от его перевозки. На каждое место  можно разместить контейнер, если его вес  где  и  – заданные величины.

Необходимо выбрать  контейнеров из  имеющихся таким образом, чтобы доход от перевозки был максимально возможным.

Математическая модель задачи выглядит следующим образом:

    

где  – неизвестные (номера выбранных контейнеров), которые требуется найти.

Решением задачи будет вектор . Каждый элемент этого вектора может принимать целое значение из отрезка  и при этом все значения  должны быть разными.

39: На рис. изображена схема, поясняющая решение этой задачи с помощью генератора размещений. Задача имеет следующие исходные данные:

 – общее количество контейнеров;

 – количество свободных мест на палубе судна;

– вес контейнеров 

 – доход от перевозки контейнеров 

 – минимальный вес контейнеров (

 – максимальный вес контейнеров 

Слева на схеме показана таблица, содержащая все размещения по три элемента множества  Эти размещения могут быть получены с помощью генератора размещений.

Количество различных способов разместить контейнеры на палубе равно:



Элементы размещений, полученных с помощью генератора используются в качестве индексов в массивах, содержащих вес и доход от транспортировки контейнеров.

Строки таблицы, представленной на схеме справа, содержат планы всех размещений контейнеров по свободным местам на палубе судна. При этом в первой заголовочной строке таблицы указаны ограничения на вес контейнера для соответствующего места (два числа, обозначающие минимальный и максимальный вес), а в остальных ячейках для каждого места приведены вес соответствующего контейнера и доход от его перевозки в формате . Закрашенные ячейки таблицы обозначают нарушение ограничений по весу контейнеров для соответствующих мест. План размещения, имеющий хотя бы одну закрашенную ячейку, является недопустимым.

На схеме только два плана размещения контейнеров являются допустимыми. Для этих планов вычисляется доход от перевозки выбранных контейнеров. Доход указан в овальной рамке рядом со строкой плана. Оптимальный план размещения контейнеров на палубе судна выделен рамкой.

40-45: На рис. представлен пример реализации на языке С++ функции **boat\_c**, решающей задачу о размещении контейнеров.

Функция имеет семь параметров, определяющих условие задачи: **m** (количество мест для установки контейнеров), **minv** (массив размерностью **m**, содержащий минимальный вес контейнера для каждого места), **maxv** (массив размерностью **m**, содержащий максимальный вес контейнера для каждого места), **n** (количество контейнеров), **v** (массив размерностью **n**, содержащий вес контейнеров), **c** (массив размерностью **n**, содержащий величину дохода от перевозки каждого контейнера), а также один возвращаемый параметр **r** (массив размерностью **m**,содержащий номера выбранных контейнеров). Если решение задачи существует, то функция возвращает положительное значение, равное величине дохода, иначе возвращается нуль.

В функции **boat\_c** применяется генератор размещений (тип **combi::accomodation**). Конструктору генератора передается два параметра: **n** (количество контейнеров) и **m** (количество мест для установки контейнеров).

Кроме того, функция вызывает три вспомогательные функции.

Функция **boatfnc::compv** позволяет проверить допустимость размещения контейнеров. Если вес всех контейнеров в размещении удовлетворяет заданным ограничениям, то функция возвращает **true**, иначе возвращается **false**.

Функция **boatfnc::calcc** дает возможность вычислить доход от перевозки контейнеров при заданном размещении.

Функция **boatfnc::copycomb** предназначена для сохранения (копирования) наиболее доходного на данном шаге допустимого размещения.

Функция **boat\_c** в цикле генерирует все возможные размещения контейнеров (функции **getfirst** и **getnext** генератора размещений), проверяет их допустимость (функция **boatfnc::compv**), вычисляет для допустимых размещений доход от перевозки контейнеров (функция **boatfnc::calcc**), фиксирует оптимальное размещение (функция **boatfnc::copycomb**) и возвращает доход от оптимального размещения или нуль, если решения нет.

Программа сначала подготавливает массивы данных: **v** (вес каждого контейнера), **c** (доход от перевозки каждого контейнера), **minv** (нижняя граница веса контейнера для каждого свободного места на палубе судна), **maxv** (верхняя граница веса контейнера для каждого свободного места), а затем вызывает функцию **boat\_c**.

В результате выполнения функция возвращает доход от оптимального размещения контейнеров на палубе судна, а также параметр (массив **r**), содержащий перечень номеров выбранных контейнеров.

46-48: На рис. представлена программа, позволяющая оценить продолжительность решения задачи о размещении контейнеров в зависимости от количества свободных мест на палубе судна и результат ее выполнения.

Здесь применяются функции **auxil::start** и **auxil::iget**, позволяющие сгенерировать случайные последовательности целых чисел в заданных пределах.

50: График, отражающий зависимость продолжительности решения задачи от количества свободных мест на палубе судна при общем количестве контейнеров, равном 11. Вид графика вполне согласуется с оценкой  сложности алгоритма генерации размещений  из 11 элементов.

Сложность .

**Лекция 4**

**Общие принципы решения задач оптимизации методом ветвей и границ**

***3: Метод ветвей и границ*** – это общий алгоритмический метод решения задач комбинаторной оптимизации. По существу, метод является вариацией полного перебора с отсевом подмножеств допустимых решений, заведомо не содержащих оптимальных решений. В основе метода лежат две процедуры: процедура ветвления (*BR*), позволяющая разбивать множество допустимых решений на непересекающиеся подмножества, и процедура вычисления нижней или верхней границы (*EV*).

Впервые метод ветвей и границ был предложен А. Лендом и А. Дойгом в 1960 г. для решения общей задачи целочисленного линейного программирования.

К сожалению, далеко не для всех задач комбинаторной оптимизации найдены процедуры ветвления и оценки, позволяющие их решить методом ветвей и границ.

4: **Решение задачи коммивояжера**

Задача коммивояжера является классическим примером задачи комбинаторной оптимизации, которая может быть решена методом ветвей и границ. Алгоритм решения, рассмотренный ниже, предложен Дж. Литлом, К. Мурти, Д. Суини и К. Кэрол в 1963 г.

Опишем процедуру ветвления **BR**, применяемую в этом алгоритме. Пусть полный взвешенный ориентированный граф  с весовой функцией  моделирует города (множество вершин *V*) и расстояния между ними (взвешенные дуги *E*) в задаче коммивояжера. Тогда решение этой задачи сводится к отысканию кольцевого маршрута  проходящего через все вершины графа и имеющего минимальную сумму весов дуг, составляющих кольцевой маршрут (кратчайший кольцевой маршрут).

Пусть  – множество всех кольцевых маршрутов, проходящих через все вершины (без повторений) графа. Очевидно, что множество  соответствует множеству всех допустимых решений задачи коммивояжера.

5: Процедура **EV** вычисления нижней границы основывается на двух утверждениях:

Утверждение 1. Изменение всех элементов строки матрицы расстояний на одно и то же число не влияет на выбор оптимального маршрута коммивояжера.

Утверждение 2. Изменение всех элементов столбца матрицы расстояний на одно и то же число не влияет на выбор оптимального маршрута коммивояжера.

Если последовательно для каждой вершины  графа  вычислить значения  и , то величина  будет составляющей веса кратчайшего кольцевого маршрута, но никогда его не превысит. Другими словами,  можно использовать как нижнюю границу веса кратчайшего кольцевого маршрута.

6: Приведем пример решения задачи коммивояжера методом ветвей и границ.

Для простоты города пронумеруем и будем их обозначать с помощью номера. Таблица содержит расстояния между каждой парой из пяти городов.

Например, расстояние из города 2 в город 3 равно 4 единицам, а из города 4 в город 5 – 8 единицам. По диагонали таблицы установлены символы, обозначающие бесконечность, что свидетельствует о невозможности такого передвижения коммивояжера.

7: В соответствии с утверждением 1: если все элементы первой строки таблицы уменьшить на 4 (наименьшее значение в строке), то это не повлияет на порядок городов в кратчайшем кольцевом маршруте, проходящем через все города по одному разу, а лишь сократит его длину на 4. Будем называть эту операцию ***приведением таблицы по строке***, а число 4 – ***константой приведения***. Аналогично можно поступить со всеми строками таблицы. На рис. *а* изображена таблица до приведения ее по всем строкам.

Столбец, помеченный символом , содержит константы приведения для каждой строки, а под столбцом в окружности указана сумма этих констант (число 13).

В соответствии с утверждением 2: если все элементы второго столбца таблицы на рис. *б* уменьшить на 1 (наименьшее значение в столбце), то это не повлияет на порядок городов в кратчайшем кольцевом маршруте, проходящем через все города по одному разу, а лишь сократит его длину на 1. Будем называть эту операцию ***приведением таблицы по столбцу***, а число 1 – ***константой приведения***. Аналогично можно поступить со всеми столбцами таблицы на рис. *б*. Столбец, помеченный символом  содержит константы приведения для каждого столбца, а в окружности указана сумма этих констант (число 1).

8: Утв.3 и 4.

Общая сумма констант приведения равна  Эту величину можно принять в качестве нижней границы длины кратчайшего кольцевого маршрута, проходящего через все города.

На рис. изображена корневая вершина дерева **T**, соответствующая множеству  – всех допустимых решений данной задачи (множеству всех кольцевых маршрутов, проходящих через все города по одному разу).

9: Таблица на рис. является таблицей, приведенной по строкам и по столбцам. Далее будем называть такие таблицы ***полностью приведенными таблицами***.

Прежде чем будет применена процедура ветвления **BR**, оценим, как будет изменяться нижняя граница подмножеств, полученных после применения процедуры **BR**  в зависимости от выбора дуги, которая используется для разбиения. Наибольший интерес представляет та дуга, которая наиболее сильно может повлиять на нижнюю границу длины допустимых кольцевых маршрутов.

Анализировать, прежде всего, следует дуги, имеющие нулевой вес, так как только их удаление может повлиять на нижнюю границу. Удаление дуги может быть промоделировано установкой ее веса равного бесконечности.

На рис. изображена таблица расстояний между городами, по которой невозможно построить конечный кольцевой маршрут, содержащий дугу (1, 4), т.е. если считать, что среди допустимых кольцевых маршрутов нет дуги (1, 4), то нижняя граница увеличится на 4. На рис. аналогично исследуются все дуги таблицы, имеющие нулевую длину.

Вычисления показывают, что удаление дуги (1, 4) позволяет получить самую большую сумму констант приведения (4), а значит, выбор этой дуги для ветвления с помощью процедуры **BR** даст самое большое увеличение нижней границы длины кольцевых маршрутов. Или проще, все допустимые кольцевые маршруты , которые могут быть построены по таблице на рис. не будут включать путь из города 1 в город 4, а длина этих кольцевых маршрутов не будет меньше, чем нижняя граница, построенная для  ***полностью приведенной таблицы*** увеличенная на 4.

10 : На рис. изображен фрагмент графа **T**, который может быть построен на этом этапе решения задачи.

Вторая ветвь решения, полученная процедурой **BR**,соответствует множеству допустимых кольцевых маршрутов, которые включают дугу (1, 4). Если дуга (1, 4) включается в кольцевой маршрут, то по всей видимости, следует этот факт запомнить и построить новую таблицу, в которой будет отсутствовать первая строка (во всех допустимых маршрутах из города 1 есть дуга только в город 4) и четвертый столбец (в город 4 есть единственная дуга и это дуга из города 1). Кроме того, очевидно, что допустимый кольцевой маршрут, содержащий дугу (1, 4), не может содержать дугу (4, 1), так как наличие этих двух дуг в кольцевом маршруте делает его недопустимым. Из этого следует, что на пересечении четверной строки и первого столбца надо поставить символ бесконечности.

11: На рис. изображена таблица, полученная из таблицы на пред. рис. вычеркиванием из нее первой строки, четвертого столбца и заменой значения в первом столбце четвертой строки на символ бесконечности. Сумма констант приведения, подсчитанная для этой таблицы, равна 1. На рис. *б* изображена таблица, которая получена после приведения таблицы на рис. 8.4, *а*.

Также изображен фрагмент графа T, имеющий две ветви. Нижняя граница дочерних узлов, вычисляется как сумма, нижней границы родительского узла и суммы констант приведения соответствующей таблицы.

Согласно алгоритму дальнейший поиск решения следует осуществлять во множестве  поскольку именно здесь на текущий момент нижняя граница является меньшей.

Проанализировав таблицу на рис., несложно вычислить, что удаление дуги (4, 3) позволит получить наибольшую сумму констант приведения (4).

12: На рис. изображена таблица, полученная из таблицы на рис. заменой нулевого значения в четвертой строке третьего столбца на символ бесконечности. Сумма констант приведения для этой таблицы равна 4. На рис. представлена таблица, полученная из таблицы на рис. путем полного ее приведения.

Таблица на рис. получена путем вычеркивания четвертой строки и третьего столбца. Сумма констант приведения этой таблицы равна 2.

13: На рис. изображен фрагмент графа **T**. Две добавленные ветви соответствуют множествам  (множество допустимых кольцевых маршрутов, содержащих дуги (1, 4) и не содержащих дуги (4, 3)) и (множество допустимых кольцевых маршрутов, содержащих одновременно дуги (1, 4) и (4, 3)).

Дальнейший поиск решения целесообразно осуществлять во множестве  так как здесь на текущий момент нижняя граница является меньшей.

14: На рис. изображен еще один шаг построения графа **T**. Анализ таблицы на рис. позволяет выявить дугу (2, 1), удаление которой приводит к максимальной сумме констант приведения (1). Следует отметить, что удаление дуги (5, 1) тоже даст такую же сумму констант приведения. В этом случае может быть выбрана любая из дуг.

Граф **T**,изображенное на рис. 8.6, *д*, содержит три вершины, которым соответствует одинаковое значение нижней границы (18). В этом случае целесообразно продолжать наращивать ветви графа, начиная с вершин, наиболее удаленных от корня. В нашем случае была выбрана вершина  Анализ таблицы позволяет выявить два последних звена кольцевого маршрута: (3, 5) и (5, 2). Эта таблица не может быть приведена, т. е. сумма констант приведения будет равной 0.

15: На рис. изображен окончательный вид графа **T**.

Для получения окончательного решения следует расставить выбранные дуги в правильном порядке: (1, 4), (4, 3), (3, 5), (5, 2), (2, 1). Сложив расстояния, соответствующие дугам кольцевого маршрута, получим 18, что совпадает с нижней границей, приписанной последнему узлу графа **T**.

Следует обратить внимание на несколько листовых вершин дерева, которым тоже соответствует нижняя граница, равная 18. Продолжение вычислений для этих ветвей в лучшем случае приведет к получению кольцевого маршрута такой же длины (18), но никогда меньшей.

**Лекция 5**

**Рекурсивные алгоритмы**

3: Многие оптимизационные алгоритмы основаны на принципе разбиения основной задачи на подзадачи, каждая из которых повторяет основную, но входные их данные таковы, что область допустимых решений становится меньше.

***Рекурсивный алгоритм*** – это алгоритм, решающий задачу путем сведения ее к решению одной или нескольких таких же задач, но в сокращенном их варианте.

Неразрывно с понятием рекурсивного алгоритма связано понятие рекурсивной функции. Существует два определения этого понятия.

Первое определение рекурсивной функции относится к теории вычислимости и является синонимом понятия вычислимой функции, т. е. функции, для вычисления значения которой можно указать алгоритм.

Второе определение, которое и будет использоваться здесь, происходит из области теории программирования.

***Рекурсивная функция*** – это функция, которая вызывает саму себя.

4: Рекурсивный алгоритм может быть записан в виде рекурсивной функции. Классическими примерами рекурсивных функций являются функции для вычисления факториала, чисел Фибоначчи и наибольшего общего делителя с помощью алгоритма Эвклида.

Рекурсивную функцию всегда можно преобразовать в цикл, и, наоборот любой цикл можно представить в виде рекурсивной функции. На рис. 2 приведен пример функции, вычисляющей факториал числа с помощью цикла.

Рекурсивная запись алгоритма, как правило, не дает выигрыша в скорости его работы. Скорее наоборот, так как вызов любой функции связан с сохранением и восстановлением контекста вызывающей функции, что является затратной по времени операцией. Кроме того, для хранения контекста операционной системой резервируется специальная секция памяти, называемая системным стеком. Если цепочка вызовов функций является длинной (иногда говорят о большой ***глубине рекурсии***), то это может привести к переполнению стека. Например, при вычислении факториала числа 25 глубина рекурсии достигает значения 24.

Часто рекурсивные функции, применяемые для решения оптимизационных задач, используют более одного рекурсивного вызова, каждый из которых работает приблизительно с половиной входных данных. Такую схему решения называют ***«разделяй и властвуй»***.

5: **Решение задачи о рюкзаке**

В предыдущих лекциях была приведена упрощенная задача о рюкзаке. Здесь эта задача будет сформулирована в классическом виде и решена с помощью рекурсивного алгоритма.

Классическая формулировка задачи допускает выбор одинаковых предметов при размещении их в рюкзаке. Математическая модель классической задачи о рюкзаке выглядит следующим образом:

   

где  – неизвестные (количество предметов каждого типа), которые требуется найти;  – объемы предметов; – стоимость единицы объема каждого предмета;  – вместимость рюкзака;.

Решением задачи при такой постановке будет вектор  Каждый элемент  вектора может принимать целое значение из отрезка  При этом, если  то -й предмет не выбран, и если  то -х предметов в рюкзак помещено  единиц.

6: Последовательность рекуррентных соотношений, позволяющая вычислить значение максимальной стоимости рюкзака равна

, ,

, ,

,

, …, , .

7: Схема решения задачи о рюкзаке с помощью рекурсивного алгоритма при        .

Все вершины дерева, кроме корневой, изображают этапы решения. Вершины помечены двумя числами: первое число – текущая стоимость рюкзака, второе – остаток неиспользованного объема рюкзака. Все этапы образуют три слоя, что определяет глубину рекурсии. Каждой вершине (этапу) соответствует вызов рекурсивной функции.

Вершины соединены дугами, указывающими связь между этапами решения. Каждая дуга имеет метку, обозначающую предположение, при котором решается очередной этап.

Для примера, рассмотрим маршрут, образованный дугами с метками   и 

Первый этап в этом маршруте отмечен меткой  означающей, что при размещении в рюкзаке двух предметов с номером 3 стоимость рюкзака станет  единиц, и при этом в рюкзаке останется  единиц объема.

Второй этап имеет метку . Решение на этом этапе осуществляется в предположении, что в рюкзаке два предмета с номером 3 и один предмет с номером 2. Поэтому  и .

На третьем этапе завершается формирование одного из допустимых решений. Неиспользованный остаток объема в  единиц позволяет поместить только один предмет первого типа. В окончательном решении, соответствующем этому маршруту, стоимость предметов, уложенных в рюкзак, равна 

Несложно заметить, что разобранный маршрут не соответствует оптимальному решению. Оптимальным будет решение , а соответствующая ему стоимость –  Этапы этого решения обозначены закрашенными овалами.

Из схемы видно, что в рекурсивном решении, как и в случае с использованием генератора, осуществляется полный перебор допустимых решений.

8: **Вычисление дистанции Левенштейна**

***Дистанция Левенштейна (расстояние Левенштейна, редакционное расстояние, дистанция редактирования)*** определяется между двумя строками и равна минимальному количеству операций вставки одного символа, удаления одного символа и замены одного символа на другой, необходимых для превращения одной строки в другую.

Впервые такой способ вычисления дистанции (меры различия) между двумя двоичными последовательностями предложил советский ученый-математик В. И. Левенштейн (род. 1935). Впоследствии способ распространился на последовательности произвольного алфавита.

Расстояние Левенштейна активно применяется для исправления ошибок в поисковых системах, в текстовых редакторах, а также в биоинформатике.

Пусть и  – две символьные строки, тогда для вычисления дистанции Левенштейна  между ними может быть использовано следующее рекуррентное соотношение:



 – количество символов в заданной строке. Например, 

 – заданная строка без последнего символа. Например, 

 – последний символ заданной строки. Например, 

Поясним принцип применения этого рекуррентного соотношения на следующем примере.

9-12: Пусть необходимо вычислить  Тогда имеем следующую последовательность шагов:

Шаги вычисления с 1 по 14 соответствуют рекурсивному погружению, а шаги с 15 по 28 – восходящему вычислению. Нетрудно убедиться, что для превращения слова «сор» в слово «спорт» достаточно удалить (или вставить) две буквы.

13-14: Представлена рекурсивная функция **levenshtein\_r**, вычисляющая дистанцию Левенштейна для двух строк, пример программы, вызывающей эту функцию и результат ее выполнения соответственно.

Функция **levenshtein\_r** имеет четыре параметра: **lx** (длина первой строки), **x** (массив символов размерностью **lx**, содержащий символы первой строки), **ly** (длина второй строки), **y** (массив символов размерностью **ly**, содержащий символы второй строки). Фунция возвращает дистанцию Левенштейна между двуми заданными строками.

Следует отметить, что в информатике часто используется ***дистанция Дамерау – Левенштейна***, которая является модификацией дистанции Левенштейна и отличается применением еще одной операции преобразования – перестановки соседних символов.

15: **Решение задачи о расстановке скобок при перемножении матриц**

Вычислим количество операций умножения, которые необходимо выполнить при перемножении трех матриц   и  размерностью соответственно   и 

Заметим, что операция умножения матриц ассоциативна и следовательно, 

Несложно подсчитать, что количество операций умножения, которые необходимо выполнить при вычислении произведения  будет  а при вычислении  –  т.е. в 10 раз больше. Таким образом, расстановка скобок при перемножении большого количества матриц может существенно повлиять на трудоемкость вычисления результата.

Рекуррентное соотношение, позволяющее вычислить минимальное количество операций умножения:



16: Определим для примера минимальное количество операций умножения для матриц   и  из рассмотренного выше примера:



  



  





17-19: Представлена функция **OptimalM**, реализующая алгоритм поиска оптимальной расстановки скобок при перемножении нескольких матриц.

Функция **OptimalM** имеет четыре входных параметра: **i** (номер первой матрицы), **j** (номер последней матрицы), **n** (общее количество перемножаемых матриц), **c** (массив размерностбю **n**, содержащий размерности матриц), а также один выходной параметр **s** (двумерный массив размерностью **n**, содержащий точки разрыва). Кроме того, к точке вызова возвращается минимальное количество операций умножения, необходимых для перемножения матриц заданной размерности.

Функция **OptimalM** является рекурсивной, так как она в процессе своей работы вызывает саму себя. Дно рекурсии достигается при совпадении значений двух первых параметров функции.

20: Расстановку скобок в заданной последовательности матриц можно осуществить с помощью двумерного массива **s**, возвращаемого функцией в последнем параметре.

Поясним принцип использования этого массива для расстановки скобок при перемножении шести матриц       размерности которых заданы массивом **Mc** на рис. 7.11. Массив (матрица) **s**, который используется далее для расстановки скобок, является результатом работы программы.

Скобки расставляются по принципу «сначала внешние – затем внутренние». Найдем элемент  в матрице **s**. Он равен 3. Это означает, что точка разрыва находится между первой и шестой матрицей после третьей матрицы, что позволяет расставить внешние скобки следующим образом: 

Точку разрыва между первой и третьей матрицей определяет элемент  а между четвертой и шестой –  После расстановки скобок выражение будет выглядеть следующим образом  Полученная расстановка скобок позволяет получить минимальное количество операций умножения, равное 15 125.

21: **Решение задачи вычисления длины наибольшей общей подпоследовательности**

Пусть  – последовательность  символов . Последовательность  называется ***подпоследовательностью*** последовательности  если она может быть получена из последовательности  удалением ее некоторых элементов. Например, если  – последовательность символов  то последовательность  состоящая из символов  является подпоследовательностью  так как она может быть получена с помощью удаления элементов 

Подпоследовательность  называется ***общей подпоследовательностью*** последовательностей  и  если она является подпоследовательностью обеих последовательностей. Например, подпоследовательность  является общей подпоследовательностью последовательностей  и 

22: ***Длиной последовательности*** (***подпоследовательности***) будем называть количество ее элементов. Например, длина последовательности  равна 6.

Подпоследовательность  называется ***наибольшей*** ***общей подпоследовательностью*** последовательностей  и  если  имеет наибольшую длину среди всех общих подпоследовательностей. Например, общая подпоследовательность  является наибольшей для последовательностей  и  В общем случае может быть несколько наибольших подпоследовательностей. Например,  тоже является наибольшей общей подпоследовательностью для последовательностей из последнего примера.

Для обозначения наибольшей общей подпоследовательности используется сокращение **LCS** (longest common subsequence).

Рекурсивный алгоритм вычисления длины LCS  для двух последовательностей  и  основывается на трех следующих очевидных утверждениях, которые приводятся без доказательства:

1. Если  то  и  является LCS для  и 
2. Если  и  то  является LCS для  и 
3. Если  и  то  является LCS для  и 

Обозначив через  длину LCS для последовательностей  и  можем записать следующее рекуррентное соотношение:



23: Рассмотрим пример вычисления длины LCS для последовательностей  и  Вычисление осуществляется по шагам. Все шаги вычисления можно разбить на две группы: с 1 по 17 и с 18 по 26. Первая группа соответствует рекурсивному погружению, вторая – восходящему вычислению. Результатом вычисления является значение  равное длине LCS. Нетрудно убедиться, что для последовательностей  и существуют две LCS:  и  имеющих длину 3.

24-25: На рис. представлена рекурсивная функция **lcs**, вычисляющая длину LCS для двух заданных последовательностей, пример программы, вызывающей эту функцию, и результат ее выполнения.

Функция **lcs** имеет четыре параметра: **lenx** (длина первой последовательности), **x** (массив, содержащий символы первой последовательности), **leny** (длина второй последовательности), **x** (массив, содержащий символы второй последовательности). Функция возвращает длину LCS заданных последовательностей.

Функция **lcs** фактически повторяет запись рекуррентного соотношения, записанного выше, и поэтому не требует дополнительного пояснения. Обратите внимание, что исходные последовательности и результат совпадают с последовательностями и результатом вычисления предыдущего примера.

**Лекция 6**

**Динамическое программирование**

1-2: Как уже отмечалось, в некоторых случаях рекурсивное решение может оказаться очень трудоемким, так как в процессе его выполнения одна и та же задача может решаться несколько раз. Простой анализ примера вычисления дистанции Левенштейна позволяет выявить такие повторные вычисления. Например, функция  вычислялась в шагах 7, 8, 7, 16 алгоритма.

Если в процессе выполнения рекурсивного алгоритма одна и та же задача решается несколько раз, то говорят, что алгоритм содержит ***перекрывающиеся подзадачи***. Например, задача вычисления функции , рассмотренная выше, является перекрывающейся подзадачей задачи вычисления функции 

3: Метод решения задачи оптимизации, реализующей рекурсивный алгоритм с перекрывающимися подзадачами, в котором каждая такая подзадача решается один раз, а ее результат сохраняется для последующего применения, называется ***динамическим программированием***.

4: Пример реализации алгоритма вычисления члена последовательности Фибоначчи методом динамического программирования.

Обратите внимание, что новой реализацией функции **fib** для хранения промежуточных значений используется статический массив. Применение массива позволяете избежать повторных рекурсивных вычислений, которые были в реализации, представленной в пред. лекции.

Заметим, что задачи вычисления факториала и наибольшего общего делителя не имеет смысла решать методом динамического программирования, так как при их рекурсивном решении не образуются перекрывающиеся подзадачи и, следовательно, нет необходимости запоминать промежуточные вычисления.

Следует отметить, что рассматриваемый здесь метод является частью более общей теории динамического программирования, основы которой разработаны Р. Беллманом. Эта теория исследует процесс пошагового решения задач оптимизации, в котором на каждом шаге из множества допустимых решений выбирается одно, оптимизирующее заданную целевую функцию.

5: **Решение задачи о рюкзаке**

Применение динамического программирования при решении оптимизационных задач обычно предполагает создание специальных таблиц для хранения промежуточных результатов.

На рис. изображены таблицы, которые используются для решения задачи о рюкзаке методом динамического программирования. Векторы, определяющие размеры () и стоимости () типов предметов, а также величина, характеризующая вместимость рюкзака (), заданы в верхней части рисунка.

Количество таблиц, которое необходимо построить для решения поставленной задачи, соответствует заданному в задаче  – количеству типов предметов.

Таблица Т.0 описывает все возможные способы размещения в рюкзаке предметов, имеющих объем 85 и стоимость 150 за единицу веса, а таблица T.1 – все способы размещения предметов объемом 50 и стоимостью 25.

Все таблицы имеют одинаковую структуру и содержат по три столбца: неиспользованный объем места в рюкзаке, объем, занятый соответствующим типом, стоимость вещей в рюкзаке.

Например, первая строка таблицы T.0 соответствует случаю, когда в рюкзак не помещено ни одного предмета этого типа. Третья строка описывает случай, когда в пустой рюкзак помещено два предмета, которые заняли объем  Третий столбец третей стоки содержит стоимость двух предметов, равную  =  единиц.

Десятая строка таблицы T.1 соответствует случаю, когда в рюкзаке остался незанятый объем, равный 215, и туда поместили два предмета, имеющих объем по 50 единиц. При этом общая стоимость вещей в рюкзаке вычисляется как сумма стоимости предметов, размещенных ранее (таблица T.0) в объеме  и предметов, помещенных в данный момент.

Определить стоимость ранее размещенных предметов можно по таблице T.0. Для этого необходимо найти строку, которая соответствует остатку 215. Это вторая строка, так как она описывает размещение в пустом рюкзаке (300 единиц незанятого объема) одного предмета объемом 85 и стоимостью 12 750. Таким образом, стоимость предметов в рюкзаке, которая будет записана в десятой строке таблицы T.1, будет  

Построение таблиц осуществляется последовательно в порядке их нумерации. Каждая таблица строится на основе предшествующей. Таблица, помеченная символом S, имеет чисто техническое назначение и предназначена для построения таблицы T.0. Таблица S состоит из одной строки, в первом столбце этой строки указывается вместимость рюкзака, а два другие столбца заполняются нулями.

В последней таблице (в нашем случае в таблице T.6) каждая строка соответствует одной из допустимых комбинаций предметов, помещенных в рюкзак. Для выбора комбинации, имеющей наибольшую стоимость, необходимо просмотреть всю таблицу и выбрать строку, в которой третий столбец имеет максимальное содержимое (на рис. эта строка выделена, содержимое третьего столбца – 39 650). Далее следует восстановить путь построения этой строки, продвигаясь от последней таблицы к первой, используя значение третьего столбца в качестве ключа.

Для того чтобы определить строку в таблице T.5, необходимо вычислить значение ее третьего столбца. Оно равно разнице содержимого третьего столбца выделенной в таблице T.6 строки (39 650) и стоимости предметов, общий объем которых является значением второго столбца (это значение равно 0). Таким образом, разница состовлять  Искомая в таблице T.5 будет иметь значение 39 650 в третьем столбце.

Отыскав в таблице T.5 необходимую строку, следует подобным способом дойти до таблицы T.0 (на рис. 7.22 все выбранные строки выделены). Разделив содержимое выделенных строк на объем соответствующего таблице типа предмета, получим их количество. Таким образом, решением данной задачи будет вектор  каждая компонента которого – количество предметов соответствующего типа.

6-7: **Вычисление дистанции Левенштейна**

8-12: На рис. представлена функция **levenshtein**, реализующая алгоритм динамического программирования вычисления дистанции Левенштейна.

Функция **levenshtein** имеет тот же набор параметров, что и функция **levenshtein\_r**,описанная в пред. лекции. Более того, реализации этих двух функций практически одинаковы. Основное отличие – применение в функции **levenshtein** массива **d** (для работы с ним используется макрос **DD**) для хранения результатов промежуточных вычислений.

Интерес представляет сравнительный анализ скорости вычисления дистанции Левенштейна функциями **levenshtein\_r** (рекурсивный алгоритм) и **levenshtein** (алгоритм динамического программирования) сделанный для строк различной длины с помощью программы, представленной на рис. 13. В программе поочередно вызываются функции **levenshtein\_r** и **levenshtein** с одинаковыми значениями параметров и замеряется продолжительность выполнения этих функций.

Рис. демонстрирует подавляющее превосходство алгоритма, основанного на динамическом программировании. При последнем просчете, в котором вычислялась дистанция Левенштейна между строками длиной 12 и 14 символов, рекурсивный алгоритм затратил на вычислении более 2 мин, а алгоритм, основанный на динамическом программировании, – менее 0,001 с.

13-18: **Решение задачи о расстановке скобок при перемножении матриц**

На рис. представлена функция **OptimalMD**, которая реализует алгоритм динамического программирования для решения задачи о расстановке скобок.

Функция **OptimalMD** имеет три параметра: **n** (общее количество перемножаемых матриц), **M** (массив размерностью **n**, содержащий размерности матриц), а также выходной параметр **s** (двумерный массив размерностью **n**, содержащий точки разрыва).

Структура функции **OptimalMD** практически идентична функции **OptimalM**, описанной в подразделе 7.1.2. Главное отличие – применение массива **M** (работа с ним осуществляется с помощью макроса **OPTIMAL\_M**) для хранения промежуточных вычислений.

19: **Решение задачи о наибольшей общей подпоследовательности**

На рис. представлена функция **lcsd**, реализующая алгоритм динамического программирования для решения задачи о наибольшей общей подпоследовательности (LCS).

Функция **lcsd** имеет три параметра: **x** (символьная строка, интерпретируемая как первая заданная последовательность), **y** (символьная строка, интерпретируемая как вторая заданная последовательность) и возвращаемый параметр **z** (символьная строка, интерпретируемая как LCS двух последовательностей, заданных первыми двумя параметрами). Функция возвращает длину LCS.

Для построения наибольшей общей подпоследовательности в функции **lcsd** используются два массива **С** и **B**.Массивы моделирует матрицы размерностью  где  – размерности соответственно первой и второй последовательностей. Номера строк матриц, начиная с первой, соответствуют номерам элементов первой последовательности, а номера столбцов, тоже начиная с первого, – номерам элементов второй последовательности. Дляудобства работы с массивами как с матрицами применяются два макроса: **LCS\_C** и **LCS\_B**.

Массив **C** предназначен для вычисления длины LCS. Элементы массива **C** – числа. Вычисление элементов **C** осуществляется пошагово, начиная с северо-западного угла матрицы. В результате вычислений в последней строке последнего столбца матрицы **C** (в юго-восточном углу) находится длина LCS.

Массив **B** предназначен для построения массива LCS (возвращаемый параметр **z**). Массив **B** моделирует матрицу, элементами которой могут быть три типа символов, обозначающих направления: **TOP** (вверх), **LEFT** (налево), **LEFTTOP** (налево и вверх). Заполнение матрицы **B** осуществляетсяодновременнозаполнением матрицы **C** и в том же порядке.

На рис. изображены матрицы С и **B** после завершения работы алгоритма функции **lcsd** для двух последовательностей B, D, C, A, B, A и A, B, C, B, D, A, B.

Порядок заполнения первой матрицы:

– первая строка и первый столбец таблицы заполняются нулями;

– каждый элемент последовательно заполняется по формуле для c[i, j]:

если символы для позиции i, j совпадают, то в неё записывается значение c[i−1, j −1]+1, иначе вычисляется максимум от соседей слева и сверху.

Элемент в правом нижнем углу показывает длину наибольшей общей подпоследовательности.

Вторая матрица заполняется следующим образом: все ячейки кроме технических строк заполняются стрелками, направленными вверх.

Если если символы для позиции i, j совпадают, то стрелка меняется на лево-вверх.

Если числовое значение от соседа слева больше, чем от соседа сверху, то стрелка меняется на лево.

Подпоследовательность получается обходом стрелок с правого нижнего угла путем добавления символов в подпоследовательность при наличии стрелки «влево-вверх» во время движения.

**Лекция 7**

**Математические основы сетевого планирования**

**Основные понятия теории графов**

3: Граф – это математическая модель, с помощью которой удобно представлять бинарное отношение. Хотя теория графов получила свое развитие задолго до появления теории множеств как самостоятельной дисциплины, большое число задач теории отношений формулируются и решаются в рамках именно этой теории.

Рассмотрим множество , которое будем назвать ***множеством*** ***вершин***, и набор  упорядоченных пар , называемый ***множеством дуг***. Пара  называется ***графом***. Запись  обозначает граф с именем , состоящий из множества вершин  и множества дуг .

4: Обычно вершины графа изображаются в виде точек, а дуги в виде соединяющих эти точки стрелок.

На рис. изображен граф , где  и .

5: Две соединенные дугой вершины графа, называются ***смежными вершинами***. Например,  и  смежные вершины графа , т.к. есть дуга .

Считается, что дуга  ***выходит из вершины***  и ***входит в вершину*** . При этом говорят, что вершины  и  ***инцидентны*** дуге , а дуга  инцидентна вершинам  и . Например, вершины  и  инцидентны дуге , выходящей из  и входящей в .

Вершины  и  дуги  называют соответственно ***начальной*** и ***конечной*** вершинами этой дуги.

6: Дуги, которые выходят и входят в одну и ту же вершину, называются ***петлями***. Например, дуга  является петлей.

Вершины, не имеющие смежных, называются ***изолированными вершинами***. Например,  – изолированная вершина.

Если множество дуг  не является симметричным отношением, то такой граф называется ***ориентированным графом***.

Если множество дуг  – симметричное отношение, то соответствующий граф называется ***неориентированным*** ***графом***. Симметричность отношения предполагает, что для каждой дуги  графа найдется дуга , имеющая противоположное направление.

Для того, чтобы разгрузить рисунок при изображении неориентированного графа, принято пару противоположных дуг изображать линией без стрелок, как это изображено на рис. 1.3.

Для того, чтобы подчеркнуть, что порядок вершин в неориентированном графе не имеет значения, при обозначении пар вершин, соединенных двумя противоположными дугами, используется запись с круглыми скобками: , а сами такие пары называют ***ребрами***.

В дальнейшем нас будут интересовать только ориентированные графы, т.к. именно они используются для моделирования сетевых графиков.

***Способы представления графов***

Существует три основных способа представления графов: матрица смежности, матрица инцидентности и списки смежных вершин.

***Матрица смежности***

7: ***Матрица инцидентности*** имеет размерность *m*×*n*, где  – количество вершин, а  – количество дуг. Каждый  элемент матрицы принимает одно из трех значений:  – вершина  не инцидентна дуге ;  – дуга  выходит из вершины ; – дуга  входит в вершину . Следует обратить внимание, что петле в матрице  соответствует столбец с одной положительной единицей, что не всегда удобно при вычислениях. Например, при подсчете количества входящих в вершину дуг следует всегда учитывать, что может быть петля, которой нет соответствующей . Поэтому матрицы инцидентности применяются редко, особенно, если граф может иметь петли.

8: ***Списки смежных вершин*** представляют собой набор из  множеств, соответствующих  вершинам графа. Множество, соответ-ствующее вершине ,  либо пустое, либо содержит номера вершин, в которые входит дуга, выходящая из вершины *i*. Например, списки смежных вершин для графа на рис. 1.1 можно представить в виде следующих пяти множеств: ; ;  . Следует отметить, что этот способ является наиболее удобным при программировании большинства задач теории графов.

При программировании реальных задач теории графов, как правило, применяются матрица смежности и списки смежных вершин. При этом часто этой информации бывает недостаточно, т.к. она отражает только структуру графа. Если с вершинами и/или дугами графа связаны какие-то дополнительные характеристики, необходимо предусмотреть возможность их хранения.

9: **Кратчайшие и максимальные пути между вершинами графа**

Пусть  – ориентированный граф, на множестве дуг которого определена функция . Функция  ставит каждой дуге  графа  в соответствие действительное число , которое часто называют ***весом дуги***. При этом саму функцию  обычно называют ***весовой функцией***, а граф – ***взвешенным графом***.

В общем случае граф  может содержать несколько путей , , …  из вершины  в вершину . ***Весом***  ***пути***  ***взвешенного графа*** *называется сумма весов дуг, составляющих этот путь: *.

***Кратчайшим путем  из вершины  в вершину *** *называется путь с минимальной весом* ******. В общем случае в одном графе может быть несколько минимальных путей.

Поиск кратчайшего пути между двумя вершинами графа является одной из часто используемых в приложениях задач. Наиболее известными способами решения этой задачи являются ***алгоритмы Дейк-стры***, ***Беллмана*** *–* ***Форда*** и ***Флойда – Уоршолла***. Здесь будет рассмотрен только алгоритм Дейкстры (Эдсгер Вибе Дейкстра, известный голландский ученый-математик, 1930–2002).

10: Алгоритм Дейкстры решает задачу о кратчайших путях во взвешенном графе  с дугами неотрицательного веса , ,  из вершины  во все достижимые вершины этого графа. Алгоритм последовательно преобразовывает исходный граф  в дерево кратчайших путей . ***Дерево кратчайших путей*** – это граф , обладающий следующими свойствами:

1. ;
2.  – ориентированное дерево с корнем ;
3.  – множество достижимых вершин графа  из ;
4. для каждой вершины  путь из  в  в дереве  является кратчайшим путем в графе .

11: Алгоритм Дейкстры предполагает, что граф  задан с помощью списков смежных вершин, а также известна весовая функция , заданная на множестве *V×V*. Функция  ставит в соответствие каждой паре вершин  вес  дуги, соединяющей эти вершины, или , если такой дуги нет в графе . Кроме того, задана вершина , относительно которой определяются все кратчайшие пути.

Для пояснения работы алгоритма Дейкстры будем использовать следующие обозначения.

12:  – множество вершин графа , смежных с вершиной . По сути ,  – это списки смежных вершин графа , с помощью которых задан граф . Иначе  – это перечень всех вершин, которые являются конечными вершинами дуг с началом в .

– верхняя оценка кратчайшего пути из вершины  в вершину . В начале работы алгоритма , а для всех остальных  оценка .

 – вершина, предшествующая вершине  в дереве кратчайших путей. В начале работы алгоритма для всех  , где – специальный символ, обозначающий пустоту. После завершения работы алгоритма множество значений  позволяет построить дерево кратчайших путей.

– множество вершин . Вначале множество  пустое: . В процессе работы алгоритма  пополняется вершинами из , для которых уже определен кратчайший путь из вершины . После окончания работы алгоритма .

13:  – множество вершин . Вначале . В процессе работы алгоритма элементы этого множества – вершины, не добавленные во множество : . После окончания работы алгоритма .

 – процедура извлечения элемента из множества  с наименьшим текущим значением . При сравнении предполагается, что , где  – любое действительное число.

 – процедура релаксации, которая определена для произвольных двух вершин  графа . Релаксация состоит в следующем: значение  уменьшается до  в том случае, если значение  меньше текущего значения . При этом операция сложения предполагает, что , где  – любое действительное число или . Кроме того, если  уменьшилось, то значение  становится равным .

 – кратчайший путь из вершины  в вершину .

14: Выполнение алгоритма Дейкстры состоит из двух этапов. На первом этапе инициализируются , ,  и  значениями, описанными выше. Второй этап алгоритма опишем с помощью мнемокода, как показано на рис. Для удобства строки мнемокода пронумерованы.

15-18: Основной цикл алгоритма (строки 1–9) на рис. выполняется до тех пор, пока все вершины графа не будут извлечены из множества  (строка 1). Извлечение вершин из множества  осуществляется с помощью процедуры  (строка 3). Извлеченная вершина помещается сначала в переменную  затем во множество  (строка 4). Далее выполняется внутренний цикл (строки 5–8), в котором для всех вершин, имеющих входящие дуги с начальной вершиной , выполняется процедура релаксации.

Результатом выполнения алгоритма Дейкстры являются массивы  и , состоящие из  элементов. Массив  позволяет построить граф кратчайших путей, а каждый элемент массива  содержит вес кратчайшего пути между вершинами  и .

На рис. 1.7 приведен пример решения задачи поиска кратчайшего пути в графе с помощью алгоритма Дейкстры. На рис. 1.7, а изображен исходный граф и проинициализированные массивы , ,  и , назначение которых разъяснялось выше. В качестве меток для вершин графа используются числа от  до . Рис. 1.7, б – 1.7, е отражают процесс решения задачи. Каждый рисунок отражает состояние массивов после завершения очередной итерации основного цикла алгоритма.

В примере на рис. 1.7 осуществляется поиск кратчайших путей из вершины  до всех остальных вершин графа. По мере решения задачи, метки вершин графа перемещаются из массива  в массив . В массиве  формируется вес пути для каждой вершины, а в массиве  – список предшествующих вершин. Результат решения представлен на рис 1.7, ж и представляет собой дерево кратчайших путей. В этом дереве из вершины  до любой другой вершины графа существует единственный путь, который является кратчайшим.

При расчете временных характеристик сетевого графика, необходимо найти критический путь, определяющий минимальное время выполнения проекта. Отыскание критического, максимального, пути в графе сводится к поиску пути с самым большим весом, называемого максимальным путем в графе.

19: ***Максимальным путем  называется путь с максимальным весом ***. В общем случае, в одном графе может быть несколько максимальных путей.

Построение максимального пути ****** во взвешенном ориентированном графе  возможно, если в нем нет контуров с положительным весом. Если в графе есть такой контур, то некоторые пути могут иметь сколь угодно большой вес, т.к. каждый обход контура увеличивает вес пути на величину веса этого контура.

Будем предполагать, что граф  безконтурный и его вершины пронумерованы так, что у любой дуги  конечная вершина всегда имеет больший номер, чем начальная: . Выше рассматривалась процедура ранжирования, основанная на алгоритме Демукрона, позволяющая привести любой безконтурный граф такому виду. Пусть все вершины графа  имеют номера от  до .

Рассматриваемый алгоритм основывается на рекуррентном выражении , где  – множество начальных вершин дуг, входящих в вершину . При этом предполагается, что для всех вершин  ранга  значение .

Вычисление значений  необходимо выполнить для каждой вершины графа в порядке возрастания номера. Каждое полученное значение  представляет собой максимальный вес пути в графе  с конечной вершиной .

Цикл вычисления  сопровождается построением массива элементов , которые формируются по тому же принципу, что и в алгоритме Дейкстры. Каждый элемент  соответствует вершине графа . Значение  – вершина (или ее номер), предшествующая вершине  в пути максимального веса с конечной вершиной .

20-25: Пример решения задачи поиска максимального пути в графе.