Лабораторная работа №3

по курсу
«Информатика (организация и поиск данных)»
(3 семестр)

Варианты заданий

1. Постановка задачи

Написать программу на C++ для сравнения различных алгоритмов поиска. Сравнение алгоритмов должно производиться на одной из приведенных задач, связанных с обработкой информации. Написать краткое техническое задание (ТЗ). Выполнить реализацию. Написать для нее тесты. Реализовать пользовательский интерфейс.

Минимальные требования к программе. В программе должно быть реализовано решение одной (или нескольких) из задач, перечисленных в разделе 2. Для многих задач предусмотрены модификации, усложняющие задачу. Задача должна быть решена с помощью одного или нескольких методов организации и поиска информации, перечисленных в разделе 3. Для каждого метода существуют различные модификации, дополняющие или улучшающие возможности или характеристики метода (и усложняющие реализацию). Все задачи, методы и их модификации снабжены числовыми оценками – рейтингом, характеризующим сложность и объем реализации. При этом если студент выбирает модифицированный вариант задачи/метода, рейтинги задачи/метода и выбранных модификаций суммируются. Студент самостоятельно выбирает конфигурацию для реализации исходя из условия: суммарный рейтинг должен быть не меньше 55. Подробнее см. раздел 4. Суммарный рейтинг выбранного студентом задания непосредственно влияет на оценку, см. раздел 5.

Основные реализованные алгоритмы необходимо покрыть тестами. Программа должна позволять выбрать любой из реализованных алгоритмов поиска и запустить его на (достаточно произвольных) исходных данных. При этом должна быть возможность как автоматической, так и ручной проверки корректности работы алгоритмов (в т.ч. должна быть возможность просмотра как исходных данных, так и результата). Программа должна обладать пользовательским интерфейсом (консольным или графическим). Пользовательский интерфейс, в особенности, графический, тестировать не требуется. Программа должна предоставлять функцию измерения времени выполнения алгоритма. Должна быть функция сравнения алгоритмов – по времени выполнения на одних и тех же входных данных[1]. Программа должна предоставлять функционал по построению графиков зависимостей, либо по выгрузке необходимых данных в открытых форматах (например, csv).

2. Задача на обработку информации

По своему характеру задачи, несколько условно, разделены на группы (типы). Студент может выбрать для решения несколько задач. Если при этом выбираются несколько задач одного типа *и их решение каким-либо образом унифицируется,* применяется прибавка к

[1] Следует рассматривать три основных случая: последовательность уже отсортирована в нужном направлении; последовательность отсортирована в обратном направлении; последовательность не отсортирована. В случае деревьев имеется ввиду сравнение времени построения дерева, а не время поиска в уже построенном дереве.

рейтингу. А именно: при выборе 2-х задач одного типа к их суммарному рейтингу прибавляется 7, при выборе 3-х задач одного типа – прибавляется 15.

№	Модификация	Рейтинг	Тип
И-1.	Построение гистограммы – распределения объектов по группам	10	С
И-1.1.	Реализация обработки в стиле map-reduce	5	
И-1.2.	Базовый уровень реализации	7	
И-1.3.	Базовый+ уровень реализации	10	
И-1.4.	Продвинутый уровень реализации	20	
И-1.5.	Максимально полный уровень реализации	30	
И-2.	Обработка разреженных векторов и матриц	8	С
И-2.1.	Обработка матриц, а не векторов	2	
И-2.2.	Реализация обработки в стиле map-reduce	5	
И-2.3.	Возможность загрузки сценария обработки из файла	4	
И-3.	Приоритезация задач на основе вычисления частотных характеристик инцидентов	5	С
И-4.	Построение алфавитного указателя	15	С
И-4.1.	[обязательно] Для задания размера страницы использовать количество символов, а не слов		
И-4.2.	Учитывать разбиение на строки	2	
И-5.	Индексирование данных	7	П
И-5.1.	Поддержка поиска по диапазону значений	5	
И-5.2.	Поддержка индексирования составных атрибутов	7	
И-6.	Кеширование данных	14	П
И-6.1.	Реализация хранения данных на диске	6	
И-7.	Задача о рюкзаке		Д
И-7.1.	Однокритериальная задача	12	
И-7.2.	Однокритериальная с доп. ограничением на объем	13	
И-7.3.	Многокритериальная задача	14	
И-7.4.	С учетом формы предметов	10	
И-8.	Задача об оптимальном поселении в гостинице	10	Д
И-9.	Поиск наиболее частых подпоследовательностей	12	Д,С
И-10.	Поиск наибольших повторяющихся подвыражений	12	Д
И-10.1.	Поиск максимально свободных подвыражений (вместо просто повторяющихся)	3	
И-11.	Проверка множества слов на соответствие регулярному выражению	14	Д
И-11.1.			
И-11.2.	Разбор строки, задающей регулярное выражение	4	
И-12.	Реализация виртуальной файловой системы	20	
И-12.1.	С реализацией прав доступа	10	
И-12.2.	Поиск файлов (построение индексов)	15	
И-13.	«Крестики-нолики»	25	М
И-13.1.	На бесконечном поле	7	
И-13.2.	С графическим UI	20	
И-14.	Реализация прототипа файловой системы	30	
И-14.1.			
И-15.	Реализация прототипа менеджера памяти	35	

И-15.1.			
И-16.	Разработка менеджера пакетов		
И-16.1.			

Типы задач		
Код	Расшифровка	Описание
С	«словарь»	Задачи на ассоциативную память
П	«поиск»	Задачи на поиск данных
Д	«дерево»	Задачи, опирающиеся на использование -арных деревьев
М	«минимакс»	Минимаксные задачи

2.1. Построение гистограммы

Гистограмма – это разбиение исходного множества на заданное число подмножеств (либо на подмножества фиксированного размера, но их число заранее может быть не известно). Разбиение производится по определенному признаку (например, возраст). Множество возможных значений этого признака разбивается на элементарные подмножества (например, диапазоны в 5-10 лет). Эти элементарные множества могут быть как одинакового размера (равномерное разбиение), так и различного (неравномерное разбиение). Интерес представляет количество элементов, попадающих в каждое элементарное подмножество.

Вход:

1 Sequence – входная последовательность элементов (например, объектов Person),
2 Параметры гистограммы (диапазон значений, разбиение)

Выход:

- IDictionary (IMap): ключом является диапазон (элементарное множество), значением – количество элементов, попадающих в этот диапазон

Базовый уровень реализации: хранение количества элементов, попадающих в каждый элементарный отрезок.

Базовый+ уровень реализации: параметризация критерия (показателя), по которому строится распределение.

Продвинутый уровень реализации: хранение (статистического) описания класса, попадающего в каждый элементарный отрезок.

Максимально полная реализация: конструктор гистограмм, включая описание классов.

2.2. Обработка разреженных векторов и матриц

Разреженный вектор – это вектор (как правильно, очень большой размерности, скажем, $10^5 - 10^7$ и более), в котором подавляющее большинство элементов (~80-90% и более) равны 0. В этом случае использование стандартных массивов для хранения значений является неоправданной тратой памяти: достаточно хранить лишь ненулевые значения (и их индексы, разумеется). Данные в таком случае следует хранить в формате ключ-значение, причем ключом является индекс в векторе, а значением – собственно элемент. Потенциально работоспособными могут быть любые варианты структур данных: сортированные последовательности, деревья, хеш-таблицы.

Разреженная матрица – это матрица (как правило, очень большой размерности, скажем, $10^5 - 10^7$ и более), в которой подавляющее большинство элементов (~80-90% и более) равны 0. В этом случае справедливы те же замечания, что и в случае с разреженными

векторами, только вместе с каждым элементов нужно хранить пару индексов, а не один. В случае использования хеш-таблиц полезно написать специализированную хеш-функцию для пары числовых (целочисленных) значений.

Вход:

1 Вектор или матрица (массив или подобная структура)

Выход:

- Разреженный вектор или матрица

В задаче требуется реализовать тип данных – разреженный вектор (или матрица), используя ISortedSequence, бинарное или B/B+-дерево, IDictionary. При выборе нескольких разных структур данных следует получить их сравнительную характеристику производительности.

$$M_{n_1 \times \ldots \times n_k} = \{a_{i_1,\ldots,i_k}\} = \{(i_1, \ldots, i_k) \to a_{i_1,\ldots,i_k}\}$$

Доступ к элементу:

1. «Полный» массив – линеаризация: $i =$
2. Разреженный массив – храним элементы, отличных от значения оп умолчанию

2.3. Приоритезация задач на основе вычисления частотных характеристик инцидентов

Вход:

1 Sequence – входная последовательность инцидентов (структур типа Incident, в которой имеется список ключевых слов)
2 Sequence (еще лучше – ISet) – множество заданий (объектов типа Task, в которых есть список ключевых слов)

Выход:

- IPriorityQueue: перечень заданий (Task), приоритизированный по частоте ключевых слов, встречающихся в инцидентах (Incident)

Указание. Перечень ключевых слов может задаваться изначально или же вычисляться путем прохода по обеим последовательностям – инцидентов и заданий. Теоретически, множества ключевых слов, встречающихся в каждом из этих наборов, могут не совпадать, а в крайнем случае – даже не пересекаться (в таком случае результирующая очередь будет состоять из списка заданий с 0-м приоритетом).

Пусть инцидент – $(i, t_{k_1}, \ldots, t_{k_l}, p_i)$

Пусть задание – $(s, t_{p_1}, \ldots, t_{p_s})$

Пусть $\{i_1, \ldots, i_n\}$ – множество инцидентов, в которых упоминается тег t_n. Тогда положим, что приоритет тега: $\sigma_{t_n} = \sum_{l=1}^{n} p_{i_l}$

Тогда приоритет задания есть сумма приоритетов его тегов $\sigma_{t_{p_s}}$.

Приоритетнее оказываются задания, у которых сильнее связь с наиболее приоритетными инцидентами.

2.4. Построение алфавитного указателя

Вход:

1 Строка, состоящая из слов, разделенных пробелами
2 Размер страницы (в словах или символах)

Выход:

- коллекция (ISet, ISortedSequence, IDictionary) пар ключ-значение, где первый элемент каждой пары – это слово, встречающееся в исходной строке, а второй элемент – номер страницы, который вычисляется по алгоритму, описанному ниже. (Используемая коллекция выбирается в разделе 3.)

Алгоритм разбиения текста на страницы.

1 Исходную строку разбить на отдельные слова, используя пробелы в качестве разделителей. В результате получится список строк, в котором каждый элемент – это слово.
2 Список слов «режется» на фрагменты так, чтобы каждый фрагмент умещался на станице. Если размер страницы указан в словах, то длина каждого фрагмента должна быть не больше этой величины. Если размер страницы задан в количестве символов, то следует вычислить суммарную длину слов выделенного фрагмента (положим, что всего фрагмент содержит N слов) и прибавить величину $N - 1$ – количество пробелов, которое необходимо, что записать эти слова раздельно.
3 При делении списка слов на страницы следует применять следующее правило: первая страница может быть заполнена не более, чем на половину, а каждая десятая – не более, чем на 3/4 от заданного размера страницы.

2.5. Индексирование данных

Индекс – специальная структура (типа ассоциативная память), ускоряющая поиск информационных объектов по ключу (значению атрибутов). По техническим причинам структура обрабатываемых данных и перечень атрибутов, по которым возможно строить индексы следует зафиксировать. Это, в частности, означает, что сортировка и другие подобные с коллекцией не допустимы. Типовой вариант (фрагмент объявления):

```
class Person {
public:
 string GetFirstName();
 string GetMiddleName();
 string GetLastName();
 string GetFullName();
 string GetFIO(); // Фамилия И.О.
 int GetBirthYear();
 int GetAge(int year); // вычислить возраст на заданный момент
};
```

Вход:

1 коллекция (`Sequence` или `ISet`) сложных по структуре объектов (например, Person)
2 список атрибутов, по которым строятся индексы (включая, если необходимо, операцию сравнения)

Выход:

- набор `IDictionary<TKey,TValue>` – индексов

Составной атрибут – атрибут, составленных из нескольких других атрибутов. В контексте данной задачи подразумевается возможность построения индекса, в котором ключом является `Tuple` (кортеж, n-ка) из значений нескольких атрибутов. Например, в случае с Person, можно выбирать такие составные атрибуты:

- (FirstName,LastName,BirthYear)
- (FullName,Age(2019))[2]
- и др.

2.6. Кеширование данных

Кеш – механизм (в т.ч. аппаратный) или структура данных, обеспечивающий более быстрое получение данных (как правило, по ключу) по сравнению с выборкой из основного места хранения[3]. Принцип работы кеша такой. Изначально кеш пуст. Далее, по мере обращения к данным, он заполняется запрашиваемыми объектами (на случай повторных запросов). Если кеш переполнен, из него удаляются данные, к которым давно не было обращений. После большого количества обращений кеш заполнен только теми данными, к которым обращения происходят чаще всего (при этом содержимое кеша со временем может меняться, если меняется частота чтения данных).

Вход:

1 коллекция (`Sequence` или `ISet`) сложных по структуре объектов (например, Person)
2 размер кеша (в количестве записей)

Выход:

- содержимое кеша (`IDictionary`) после некоторого (относительно большого) количества обращений к данным

Замечание: скорость доступа к данным в кеше должна заметно отличаться от скорости доступа к данным в основном месте хранения (например: хеш-таблица vs дерево поиска, оперативная память vs HDD).

2.7. Задача о рюкзаке

Пусть имеется двухмерный рюкзак[4] (см. пример на рис. ниже) и набор предметов (каждый из которых занимает в рюкзаке сколько-то место. Каждый предмет, кроме размеров, характеризуется весом и стоимостью. Требуется найти такой вариант наполнения рюкзака, чтобы:

вариант a) суммарная стоимость предметов в рюкзаке была максимальной, а суммарный объем – не превосходил заданной величины (объема рюкзака; считается, что форма предметов игнорируется)
вариант b) суммарная стоимость предметов в рюкзаке была максимальной, а объем и масса не превосходят заданных величин

[2] Т.е. возраст по состоянию на 2019 год.
[3] Обычно существуют какие-то ограничения, препятствующие размещению всех данных в таких быстрых структурах, например, объем данных. Это может быть связано с ограниченной вместимостью кеша (например, хеш-таблица в оперативной памяти vs B-дерево на жестком диске) или высокой стоимостью (например, использование SSD vs HDD).
[4] Частный случай одной из классических задач методов оптимизации

вариант c) суммарная стоимость предметов в рюкзаке была максимальной, но ограничена грузоподъемность (суммарный вес не должен превосходить заданной величины) и следует заполнить рюкзак как можно более эффективно (ΔV[5] $= 0$ или $\Delta V \leq \varepsilon$)

вариант d) суммарная стоимость предметов в рюкзаке была максимальной, заполнение рюкзака должно быть максимальным (при ограниченной вместимости), но ограничена грузоподъемность (суммарный вес не должен превосходить заданной величины)

или

вариант e) суммарная стоимость предметов в рюкзаке была максимальной, а суммарный вес – минимальным (при ограниченной вместимости)

При этом все предметы должны полностью умещаться в рюкзаке.

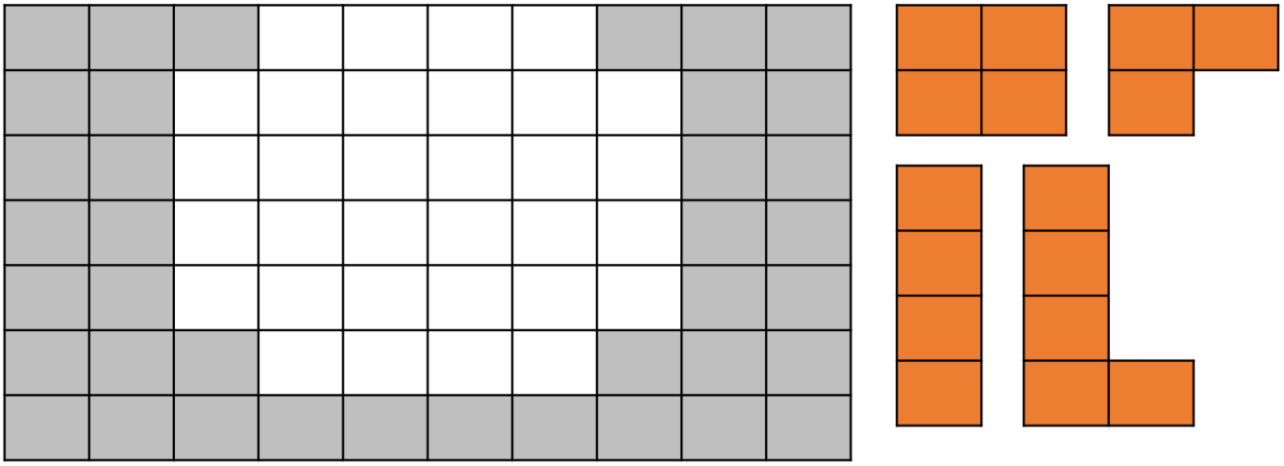


Рис. 1 – Пример форм рюкзака и предметов

Решение задачи – это последовательность, в которой предметы помещаются в рюкзак, и состояние рюкзака.

Допустимое решение – это решение, удовлетворяющее указанным выше условиям.

Вход:

1 форма, размеры рюкзака (двухмерный массив?)
2 параметры предметов (Sequence, ISet, IDictionary)
3 (для варианта a) ограничение по весу

Выход:

- набор допустимых решений

Структура решения. Решать задачу следует с помощью построения дерева решений. Каждый узел такого дерева соответствует состоянию рюкзака (какие предметы там лежат и как именно размещены). Корень такого дерева соответствует пустому рюкзаку. Пусть рюкзак частично заполнен и этому состоянию соответствует узел n. Пусть на данном этапе в рюкзак можно положить предметы $p_{i_1}, \dots, p_{i_k}$ (и никакие другие) – с учетом ограничений на вес и объем. Тогда у данного узла n может быть, по крайней мере, k дочерних узлов. Их может быть больше, если один и тот же предмет можно разместить несколькими образами. Каждое размещение порождает свой узел. Пусть для i_j-го предмета существует m_{i_j} вариантов

[5] Объем рюкзака, который остался не заполнен

размещения. Тогда всего у узла n может быть $m_{i_1} + \cdots + m_{i_k}$ дочерних узлов. Если в рюкзак нельзя положить ни одного предмета, соответствующий узел будет терминальным (листовым). Для ускорения работы алгоритма следует придерживаться общего правила: сначала нужно пытаться положить наиболее ценные и в то же время наиболее легкие и маленькие предметы.

Каждый листовой узел, таким образом, соответствует максимально заполненному рюкзаку и представляет допустимое решение. Наилучшим решением будет такой листовой узел, у которого наибольшая стоимость. Пусть от корня дерева до листового узла соответствует последовательности, в которой предметы помещались в рюкзак.

Упрощенная модификация задачи состоит в том, чтобы не учитывать форму рюкзака и предметов, а только их объем. В этом случае всегда у предмета будет лишь 1 способ размещения в рюкзаке (если его вообще можно разместить).

2.8. Задача об оптимальном поселении в гостинице

Требуется разместить группу туристов в гостинице так, чтобы суммарная стоимость была наименьшей. Номера могут быть одноместными, двухместными и трехместными. Мальчики и девочки селятся раздельно, кроме случаев супружеских пар. Кроме того, у некоторых туристов есть приятели и неприятели: приятели предпочитают селиться вместе, а неприятели категорически не будут селиться в один номер.

Вход:

1 список номеров каждого типа и их стоимость
2 список туристов и их приятелей-неприятелей-супругов
3 максимальное количество вариантов расселения, которое требуется найти

Выход:

- размещение туристов по номерам (например, `IDictionary<Room,ISet<Person>>`)

Развитие задачи состоит в том, чтобы дополнительно учитывать пожелания людей (одиночное поселение, с видом на озеро и т.п.). Это немного усложняет процедуру выбора, кого в какой номер селить, но зато значительно сокращает количество допустимых вариантов расселения.

Структура решения. Решать задачу следует с помощью построения дерева решений. Каждый узел такого дерева соответствует некоторому (частичному) размещению людей. Корневой узел соответствует варианту, когда никто никуда не поселен (номера, в которые группа может заселяться, пустые). Каждый раз, когда кто-то заселяется в номер, порождается дочерний узел. Количество свободных мест уменьшается, но и количество людей, которых осталось расселить, – тоже. Процесс расселения завершен, когда расселять больше некого (это соответствует листовому узлу в дереве). Следует руководствоваться следующим общим принципом: в первую очередь расселять супругов, затем друзей. После этого – всех остальных, избегая поселять недругов в один номер.

Довольно часто будет получаться так, что количество допустимых решений очень велико. Поэтому найти все допустимые решения может оказаться практически не выполнимым. Поэтому и вводится ограничение на количество решений (вход 3).

2.9. Поиск наиболее частых подпоследовательностей

Пусть имеется строка s (конечная последовательность символов в заданном алфавите Ω). Требуется найти такую подстроку $s' \subset s$, у которой число вхождений наибольшее (или же несколько таких подстрок). Очевидно, что:

a) Наибольшим количеством вхождений будут обладать наиболее короткие подстроки, а именно – односимвольные. Однако практически полезным обычно оказывается поиск более длинных подстрок.
b) Максимальная длина подстроки, которая может встретиться в данной строке длины n, равна $n/2$.

В базовом варианте решения данной задачи максимальную и минимальную длины искомых подстрок можно задавать параметрами.

Вход:

1 входная строка (`Sequence<char>`, вместо `char` может быть и другой тип)
2 максимальная (l_{min}) и минимальная (l_{max}) длины искомых подстрок

Выход:

– список (`Sequence`) или сортированный список (`SortedSequence`) пар: строка – число вхождений

Общая идея решения данной задачи состоит в том, чтобы построить таблицу префиксов. Таблица суффиксов может быть представлена как словарь, ключом в котором является префикс, а значением – список позиций (индексов) в исходной строке, в которых он встречается. Для целей данной задачи можно построить таблицу префиксов длинной от l_{min} до l_{max}, а затем отобрать те, у которых наибольшее число вхождений. Таблицу суффиксов можно построить за один проход вдоль исходной строки:

```
for i = 0 to length(s):
  for j = l_min to l_max:
    ss = s[i:j] // subsrt(s, i, j)
    if !(dict.hasKey(ss))
      dict.add(ss, new ArraySequence<int>)
    dict[ss].Append(i)
```

Развитием этой задачи является вариант, когда l_{min} и l_{max} не фиксируются как параметры. Вместо этого для оценки частоты используются относительные величины, например:

$$\left(\frac{l_{s'}}{l_s}\right)^2 c_{s'},$$

где l_s – длина исходной строки, $l_{s'}$ – длина рассматриваемой подстроки, $c_{s'}$ – число вхождений s' в s.

Примерный вид таблицы суффиксов:

abeedcadaedacadccba

beedcadaedacadccba

eedcadaedacadccba

edcadaedacadccba

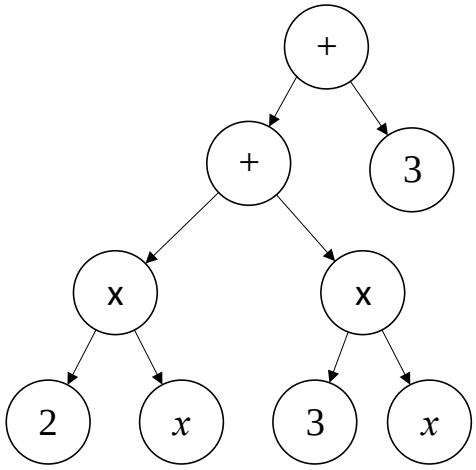
…

ba

a

2.10. Поиск наибольших повторяющихся подвыражений

Решение задачи опирается на представление выражений т.н. *абстрактными синтаксическими деревьями* (АСТ). В АСТ листовые (называемые в таком случае *терминальными*) узлы соответствуют атомарным выражениям (константам и переменным), а нелистовые (*нетерминальные*) – операциям или операторам. Пример АСТ для выражения $2x + 3y + 5$:



АСТ помогают организовывать вычисления: вычисления осуществляются в общем направлении снизу вверх (и, как правило, слева направо).

Вход:

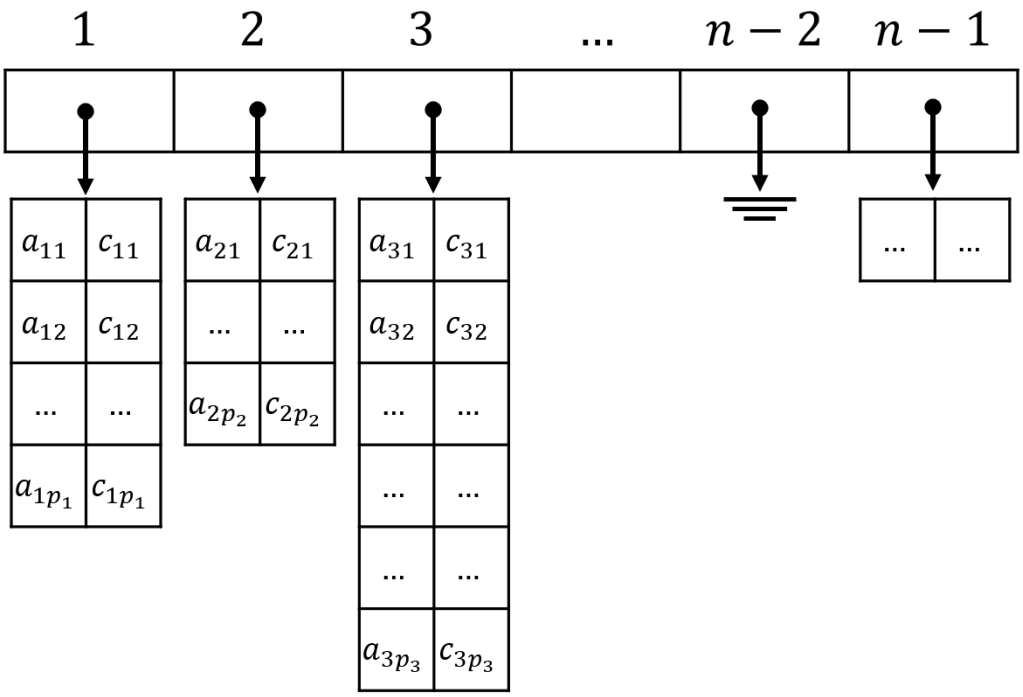
1 входное выражение в виде АСТ (или набор таких выражений)

Выход:

– список (`Sequence`) или сортированный список (`SortedSequence`) пар: подвыражение (например, указателей на корневые узлы) и количество вхождений этого подвыражения

Схема решения задачи поиска повторяющихся подвыражений состоит в том, что сначала находятся все повторяющиеся поддеревья высоты $1, \dots, n-1$ (где n – высота всего исходного дерева). Затем, если какое-либо из найденных поддеревьев полностью входит в какое-либо другое дерево (только одно или же хотя бы в одно – в зависимости от вариации задачи), то меньшее поддерево вычеркивается из списка найденных поддеревьев. Существенно, что для целей данной задачи рассматриваются только «оконечные» поддеревья, «стоящие» на терминальных узлах, т.е. соответствующие законченным подвыражениям, например, «2», «x», «$2 \times x$», и так далее. Упрощенно, можно придерживаться следующего алгоритма:

Шаг 1: Вычислить список всех поддеревьев высоты $1, \dots, n-1$. Этот список удобно представить как список списков: в каждом из вложенных списков лежат поддеревья одной и той же высоты, эти списки расположены в общем списке в порядке возрастания высоты. Элементами вложенных списков должны быть пары «поддерево-количество вхождений».



Шаг 2: Исключить из полученного списка поддеревья, встречающиеся только 1 раз. Это эквивалентно применению к списку операции where с условием $count \Rightarrow count > 1$[6].

Шаг 3: Для каждого поддерева a_i найти список поддеревьев l_{a_i}, в которые входит это поддерево a_i.

Шаг 4: Исключить поддеревья, у которых $\text{length}(l_{a_i}) = 1$ (либо $\text{length}(l_{a_i}) \geq 1$). Эквивалентный эффект получится, если применить к списку операцию where с параметром $l \Rightarrow \text{length}(l) \neq 1$ (либо $l \Rightarrow \text{length}(l) = 0$).

Шаг 5: Применить к списку списков `flatmap`.

2.10.1. Поиск максимально свободных подвыражений

Максимально свободное подвыражение (МСП) – это такое подвыражение, которое не содержит (свободных) переменных, но при этом никакое подвыражение этим свойством не обладает. Некоторые примеры:

Является МСП	Не является МСП
5	x
$5 + 6$	$x + 6$
$\lambda x. x + 6$	$\lambda x. x + y$

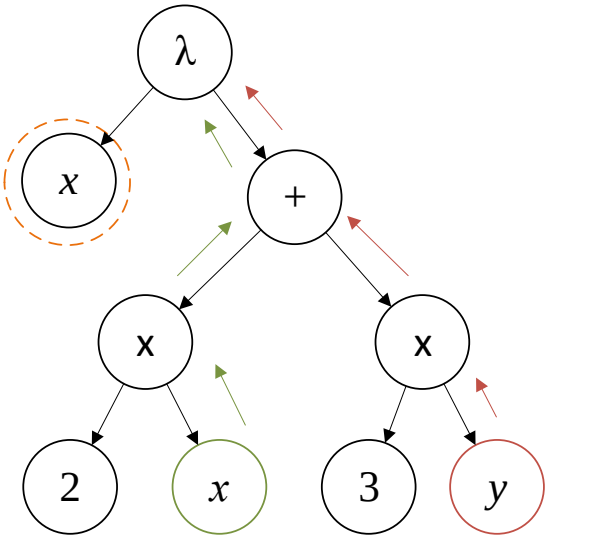
В простейшем случае, список МСП можно получить, если в предыдущем алгоритме выполнить дополнительный шаг:

[6] $x \Rightarrow x > 1$ (по другой нотации $\lambda x. x > 1$) – иной способ записи выражения C++:

```
[](auto x) => { return x > 1; }
```

Шаг 6: Исключить из полученного списка выражения, не удовлетворяющие свойству максимально свободных.

Алгоритм проверки поддерева на свойство МСП: для каждого терминального узла нужно проверить, имеется ли в числе его предков (внутри данного поддерева) узел с меткой λ:



$$\left(\lambda x.\lambda y.(\lambda x.\lambda y.2x+3y)(x+3)\left((\lambda x.x+4)(y)\right)\right)(3)(4)$$

2.11. Проверка множества слов на соответствие регулярному выражению

Вход:

1
2

Выход:

–

2.12. Реализация виртуальной файловой системы

Пусть имеется набор файлов, размещенных в реальной файловой системе (возможно, в одной выделенной папке, а возможно и в разных). Требуется реализовать средства, описывающие иерархическую структуру директорий. Исходные файлы привязываются к этой виртуальной, существующей лишь в оперативной памяти, структуре, без физического перемещения. Требуется оснастить такую виртуальную файловую систему (ФС) управляющей оболочкой:

- создание нового файла в рамках виртуальной ФС должно происходить путем прикрепления к ней существующего файла;
- удаление из виртуальной ФС не должно приводить к удалению файла из физической ФС;
- перемещение и переименование файла в виртуальной ФС не должно затрагивать физический файл.

В качестве дополнительной задачи – реализовать механизм управления правами доступа с использованием групповых политик и наследования.

2.13. «Крестики-нолики»

Вход:

1 игровая ситуация (расположение крестиков и ноликов на двухмерном поле)

Выход:

- рекомендация хода ноликами (координаты на поле)

В целом, решение строится на упорядоченном переборе возможных вариантов и построении дерева решений. Основное отличие от, например, задачи с рюкзаком – в выборе оценивающей функции. Кроме того, следует вводить искусственные ограничения на рассматриваемые варианты: например, если крестик или нолик ставится на значительном удалении от уже поставленных, то не так важно, где именно он будет поставлен. Такие ограничения существенно снижают объем вычислений.

2.14. Реализация прототипа файловой системы (семестровая задача)

Пусть имеется адресное пространство, возможно, разбитое на несколько областей – файлов, размещенных в постоянной памяти (HDD, SSD). Требуется реализовать симуляцию файловой системы:

- размещение объектов, в т.ч. с учетом разбиения пространства на страницы и сектора определенного (желательно, настраиваемого) размера;
- организацию объектов в иерархическую структуру по типу файловой системы;
- манипуляции над файловой структурой (перемещение/копирование файлов, удаление, создание, операции с папками);
- доступ к содержимому файла с помощью Stream<char>.

Крупные объекты могут фрагментироваться, размещаться в различных, в т.ч. несмежных страницах/секторах и даже областях. При реализации следует учесть, что в общем случае один файл занимает несколько секторов, а один сектор может вмещать несколько файлов. При этом сектор может читаться (в оперативную память) из адресного пространства только целиком.

2.15. Реализация прототипа менеджера памяти (семестровая задача)

Пусть имеется непрерывная область памяти. Требуется реализовать набор средств, обеспечивающих размещение в этом пространстве объектов различного размера. Размещение объекта должно выполняться перегруженным оператором `new`, а освобождение памяти – перегруженным оператором `delete`. Фрагментация объектов не допускается, т.е. выделять можно только непрерывные участки памяти. При размещении объектов следует учитывать и использовать память, которая была прежде занята, а затем освобождена оператором `delete`. Размер объектов, которые необходимо размещать, может быть произвольным.

Кроме того, следует реализовать элементы механизма автоматической сборки мусора, в первую очередь – механизм умного указателя с подсчетом числа ссылок.

2.16. Разработка менеджера пакетов

Обычно под пакетом понимается упакованная и укомплектованная структурная единица программного обеспечения, отличающаяся относительной замкнутостью. Пакет обычно включает в себя исполняемые и некоторые вспомогательные файлы, но в общем случае просто является множеством файлов (артефактов), снабженным метаданными. При этом пакет может не быть самодостаточным, а иметь некоторые внешние зависимости (другие пакеты).

К метаданным пакетам, в относительно общем случае, относится:

- версия,
- связь с другими версиями этого же пакета или иногда с другими пакетами (например, некоторые пакеты могут замещать собой другие пакеты определенных версий),
- зависимости,
- схему разворачивания (какие файлы в какие места файловой системы на целевом компьютере должны быть помещены),
- скрипты, которые необходимо выполнить до или/и после установки пакета.

Задачи, решаемые менеджером пакетов:

- построение дерева зависимостей данного пакета и проверка того, что все необходимые зависимости установлены,
- установка пакета при условии наличия необходимых зависимостей (включая исполнение скриптов),
- поиск пакета, подходящего под заданную спецификацию зависимости,
- сохранение целостности:
    - гарантия того, что пакеты не будут дублироваться,
    - гарантия того, что пакет не будет установлен, если не удовлетворены все зависимости,
    - гарантия того, что не будет нарушена работоспособность уже установленных пакетов (не будут удалены чужие зависимости[7]).

Задача на программирование: разработать прототип менеджера пакетов.

3. Структуры данных и алгоритмы поиска и их модификации

В каждой задаче требуется реализовать одну или несколько из следующих структур: множество (ISet), сортированная последовательность (ISortedSequence), очередь с приоритетами (IPriorityQueue) или ассоциативная память (IDictionary). В зависимости от того, какие из этих структур студент реализует и какими для этого пользуется «базовыми» структурами, к рейтингу формируемого задания прибавляется величина из таблицы ниже.

	Sequence	Бинарное дерево	B/B+ дерево	Хеш-таблица	IDictionary[8]
Stream	5	5			
ISet	3	5	7	5	2
ISortedSequence	3	5	7	–	2
IPriorityQueue	–	5	7	5	2
IDictionary	2	5	7	5	–

[7] Например, при установке пакетов «на замену»

[8] Суммируется с рейтингом за реализацию IDictionary

«Базовые» структуры допускают определенные вариации и усовершенствования. Указанные рейтинги пропорциональны сложности реализации. В приведенном списке элементы верхнего уровня соответствуют методу или алгоритму, а элементы нижнего уровня – это модификации. При формировании индивидуального задания рейтинг метода суммируется с реализуемыми модификациями. Модификации одного и того же метода, как правило, не являются взаимоисключающими, их рейтинги также могут суммироваться.

№	Модификация	Рейтинг
М-1.	Stream	
М-1.1.	проецируемый в память	10
М-1.2.	проецируемый в бинарный файл	12
М-1.3.	проецируемый в структурированный файл (бинарный, csv, xml, json[9])	10
М-2.	Бинарный поиск по отсортированной последовательности (базовая реализация)	10
М-2.1.	Возможность выбора пропорции деления: пополам, либо в соответствии с «золотым сечением»	5
М-2.2.	Пропорции деления задаются параметром – парой натуральных чисел, либо номером последовательности Фибоначчи	5
М-3.	Бинарное дерево	15
М-3.1.	Сбалансированное	7
М-3.2.	Прошитое	7
М-4.	B/B+-дерево (без удаления)	20
М-4.1.	Операция удаления	15
М-5.	Хеш-таблица, реализация интерфейса IDictionary<TKey,TElement> (без разрешения коллизий)	12
М-5.1.	Разрешение коллизий с помощью списков	7
М-5.2.	Разрешение коллизий с помощью смещения адресов	7
О-1	Реализовать поддержку итераторов	
О-1.1.	В Sequence	3
О-1.2.	В бинарном дереве	4
О-1.3.	В B/B+-дереве	4
О-1.4.	В хеш-таблице	3
О-2	Хранение данных в структурированном бинарном или текстовом файле	10

Пояснения.

1. Деревья поиска следует реализовывать как шаблонные классы. Это позволит как хранить в них сами данные, так и лишь индексы, отмечающие положение данных в последовательности. У шаблона один параметра-типа: тип хранимого значения.
    a. В случае, если хранятся простые объекты, например, вещественные числа, то тип этих объектов выбирается в качестве значения этого параметра: double (например, `Tree<double>`).
    b. Если дерево строится «над» числовой последовательностью и требуется хранить положение элемента (имеющего тип `T`, например, `double`, в случае вещественных чисел) в исходной последовательности, то тип хранимого значения будет парой `Pair<T,int>`. Ключом в такой паре ключ-значение

[9] Для работы с форматами json, xml рекомендуется использовать существующие библиотеки.

будет первый элемент, т.е. интересующий нас объект, а значением – его индекс (целое число `int`) в последовательности.

   c. Если последовательность состоит из сложных информационных объектов (например, класс `Student`), а поиск ведется по атрибуту типа `T`, то дерево будет инстанцировано с параметрами шаблона: `Tree<Pair<T,int>>` (в предположении, что хранимым является именно индекс объекта в последовательности), либо `Tree<T,Student*>` (если в дереве хранятся именно указатели).

2. Для хранения элементов в узлах B/B+ деревьев следует использовать реализованный ранее Sequence.
3. Алгоритм построения дерева из последовательности может быть реализован в одном из следующих вариантов:
   a. Как статический метод класса, описывающего дерево (не желательно, т.к. создает жесткую связку между классом дерева и классом последовательности).
   b. В составе отдельного класса (например, `BinaryTreeBuilder`). Это позволит в дальнейшем легко расширять функциональность этого класса, добавляя методы для построения деревьев из других структур данных (например, «голых» массивов или `std:vector<>`).
   c. В виде отдельной функции.
4. В случае с хранением данных в бинарном файле, дерево хранит лишь индексы – положение данных в файле.
5. Краткая спецификация `SortedSequence`:

Класс	`SortedSequence<TElement>`	
Является модификацией интерфейса Sequence, описывающей отсортированные последовательности. Основное отличие состоит в том, что методы Append, Prepend и InsertAt заменены одним методов Add.		
Название	**Сигнатура**	**Описание**
Атрибуты		
Length	`int GetLength()`	Длина последовательности (количество элементов)
IsEmpty	`int GetIsEmpty()`	Признак того, является ли последовательность пустой
Методы		
Get	`TElement Get(int index)`	Получение элемента по индексу.
GetFirst	`TElement GetFirst()`	Получить первый элемент последовательности
GetLast	`TElement GetLast`	Получить последний элемент последовательности
IndexOf	`int IndexOf(TElement element)`	Получить индекс элемента последовательности. Если указанного элемента в последовательности не содержится, возвращается значение -1.
GetSubsequence	`SortedSequence<TElement> GetSubsequence(int startIndex, int endIndex)`	Получить подпоследовательность: начиная с элемента с индексом startIndex и

		заканчивая элементом с индексом endIndex
Add	`void Add(TElement element)`	Добавить элемент в последовательность. Элемент автоматически вставляется так, что итоговая последовательность остается отсортированной.
GetKeys	`Sequence<TElement> GetKeys()`	Возвращает список всех ключей в словаре
GetValues	`Sequence<TElement> GetValues()`	Возвращает список значений в словаре в произвольном порядке

6. Хеш-таблица должна хранить данные в структуре данных, обеспечивающих обращение по индексу (index-based addressing) – например, Sequence. Хеш-функция должна вычислять положение в этом массиве на основании ключа. Это достигается в 2 этапа: сначала для каждого используемого типа ключа T выбирается некоторая функция $\mathrm{GetHashCode}: T \to \mathrm{int}$, которая для каждого экземпляра типа T возвращает некоторое положительное целое (int). Эта функция должна удовлетворять следующему обязательному условию:

   Если значения этой функции для некоторых двух элементов различаются, то и сами элементы различны.

   Полученное таким образом целое число является промежуточным для вычисления итогового значения хеш-функции. Хеш-функция также должна удовлетворять приведенному выше обязательному условию. Значение хеш-функции трактуется как индекс элемента в внутреннем массиве хеш-таблицы.

   В минимальном варианте необходимо реализовать хотя бы одну из следующих возможностей:

   a. использовать какую-либо списковую структуру (например, реализованный ранее тип данных Sequence) для обработки коллизий;
   b. автоматическую перестройку хеш-функции и внутренней последовательности при возникновении коллизий.

   Для получения максимального балла необходимо реализовать обе указанные возможности, а также автоматическую перестройку при удалении элементов.
7. Перестройку хеш-таблицы необходимо проводить не при каждом удалении, а лишь тогда, когда число элементов стало ниже некоторого порогового значения. Более точно, следует зафиксировать два (необязательно целых) положительных числа: p и q (причем $p \geq q > 1$). При этом хещ-таблица характеризуется двумя неотрицательными целыми числами: количеством элементов (n) и вместимостью (c). Всякий раз, когда оказывается, что $n \leq {}^{c}/_{p}$, следует сжать в q раз. Если же оказывается, что $n = c$, то таблицу следует расширить в q раз.
8. Краткая спецификация `IDictionary`:

Класс	`IDictionary<TKey,TElement>`	
Название	**Сигнатура**	**Описание**
Атрибуты		
Count	`int GetCount()`	Количество элементов,

		которое фактически хранится
Capacity	`int GetCapacity()`	Вместимость – максимальное количество элементов, которое можно поместить в таблицу без необходимости ее перестройки.
Методы		
Get	`TElement Get(TKey key)`	Получение элемента по ключу. Выбрасывает исключение, если элемента не существует.
ContainsKey	`bool ContainsKey(TKey key)`	Проверка, что в таблице уже есть элемент с заданным ключом.
Add	`void Add(TKey key, TElement element)`	Добавить элемент с заданным ключом.
Remove	`void Remove(TKey key)`	Удаляет элемент с заданным ключом. Выбрасывает исключение, если заданный ключ отсутствует в таблице.

Пример. Будем искать студентов по ФИО (FullName).

```
int GetStringHashCode(string s) { /*…*/ }
Dictionary<string,Student> students(&GetStringHashCode, 25);
students.Add(student1.GetFullName(), student1);
students.Add(student2.GetFullName(), student2);
students.Add(student2.GetFullName(), student3);
// students.ContainsKey(student1.GetFullName()) => TRUE
// students.ConstainsKey(student4.GetFullName()) => FALSE
students.Get(student1.GetFullName()) // => student1
students.ConstainsKey(student4.GetFullName()) // => Exception
students.GetCount() // => 3
students.GetCapacity() // => 25, если автоматической перестройки
                       // нет и 25/q, если есть
```

9. В зависимости от того, какие структуры выбраны для реализации, появляются дополнительные опции (O-1) – реализация итератора. Краткая спецификация `IIterator`:

Класс	`IIterator<TElement>`	
Название	**Сигнатура**	**Описание**
Атрибуты		
CurrentItem	`int GetCurrentItem()`	Получение текущего элемента по ключу. Выбрасывает исключение, если итератор находится за границами коллекции.

HasNext	`bool HasNext()`	Проверка того, итератор не достиг конца коллекции.
Методы		
Next	`bool Next()`	Перейти на следующий элемент, если возможно. Если переход удалось выполнить, возвращает true, иначе – false.
TryGetCurrentItem	`bool Add(TElement& element)`	Попытаться получить текущий элемент. Если попытка удалась, то element содержит полученное значение, а метод возвращает true; в противном случае метод возвращает false, а значение element не определено.

Пример 1 использования.

```
Sequence* seq = new ArraySequence();
// добавление элементов в seq
IIterator* it = seq->GetIterator();
for (it->Next(); it->Next(); it->HasNext())
      cout << it->GetCurrentItem();
```

Пример 2.

```
BTree btree = new BTree();
// добавление элементов в btree
 for (it->Next(); it->Next(); it->HasNext())
      cout << it->GetCurrentItem();
```

Для реализации итератора для бинарных и B/B+-деревьев достаточно реализовать обход дерева.

4. Методика формирования индивидуального задания

Формирование индивидуального задания происходит в 2 шага:

Шаг 1: Выбор решаемой задачи и ее модификаций из раздела 2. Вычисляется рейтинг задачи в баллах.

Шаг 2: Выбор реализуемых алгоритмов и структур данных и их вариантов из раздела 3. Вычисляется суммарный рейтинг выбранных структур данных и их модификаций.

> Сумма рейтингов, полученных на шагах 1 и 2, должна быть не меньше 55.

Каждая задача имеет базовую оценку T_i^0 в баллах (обычно, 1 балл). Каждая предлагаемая модификация также имеет рейтинг в баллах, пропорциональную ее сложности ($T_{i,j}$). Вычисляется суммарная оценка сложности выбранной задачи: $T_i = T_i^0 + T_{i,j}$.

Далее, из таблицы 3 выбираются подходящие структуры данных и алгоритмы. Каждый из них имеет свой рейтинг A_i^0. Далее, выбираются модификации реализуемых структур; каждая модификация имеет рейтинг $A_{i,j}$. Вычисляется суммарный рейтинг выбранных алгоритмов и структур: $\sum A_i = \sum\left(A_i^0 + \sum A_{i.j}\right)$.

Задача и алгоритмы для ее решения должны быть выбраны так, чтобы получившийся суммарный рейтинг $T_i + \sum A_i$ был ***не*** **меньше** 55.

Следует учитывать, что в некоторых задачах структуры данных, которые нужно использовать, частично зафиксированы условием.

5. Критерии оценки

1.	Качество программного кода:	– стиль (в т.ч.: имена, отступы и проч.) (0-2) – структурированность (напр. декомпозиция сложных функций на более простые) (0-2) – качество основных и второстепенных алгоритмов (напр. обработка граничных случаев и некорректных исходных данных и т.п.) (0-3)	0-7 баллов
2.	Качество пользовательского интерфейса:	– предоставляемые им возможности (0-2) – наличие ручного/автоматического ввода исходных данных (0-2) – настройка параметров для автоматического режима отображение исходных данных и промежуточных и конечных результатов и др. (0-2)	0-6 баллов
3.	Качество тестов	– степень покрытия	0-3 баллов

		– читаемость – качество проверки (граничные и некорректные значения, и др.)	
4.	Объем выполненного задания	Оценивается объем выполненного задания, исходя из суммарного рейтинга: менее 46 — 0 баллов 46 – 48 — 1 балл 49 – 51 — 2 балла 52 – 54 — 3-4 балла 55 и более — 5 баллов	0-5 баллов
5.	Владение теорией	знание алгоритмов, области их применимости, умение сравнивать с аналогами, оценить сложность, корректность реализации	0-3 баллов
6.	Оригинальность реализации	оцениваются отличительные особенности конкретной реализации – например, общность структур данных, наличие продвинутых графических средств, средств ввода-вывода, интеграции с внешними системами и др.	0-9 баллов
		Итого	0-30 баллов

Для получения зачета за выполнения лабораторной работы необходимо соблюдение всех перечисленных условий:

- оценка за п. 1 должна быть не менее 3 баллов
- оценка за п. 4 должна быть не менее 3 баллов
- оценка за п. 5 должна быть больше 0
- суммарная оценка за работу без учета п. 6 должна быть не менее 15 баллов